

RL-TR-95-295, Vol II (of four)
Final Technical Report
April 1996



ROMULUS, A COMPUTER SECURITY PROPERTIES MODELING ENVIRONMENT: ROMULUS THEORIES

Odyssey Research Associates, Inc.

S. Brackin, S. Foley, L. Gong, B. Hartman, A. Heff, G. Hird,
D. Long, D. McCullough, I. Meisels, D. Rosenthal,
I. Sutherland, and A. Weitzman

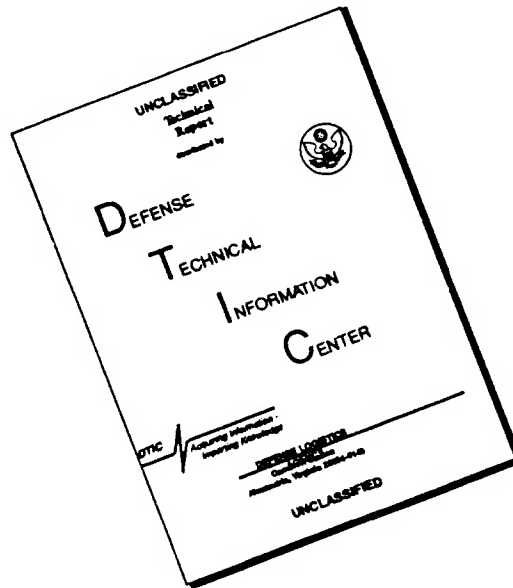
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

19960724 066

DTIC QUALITY INSPECTED 3

**Rome Laboratory
Air Force Materiel Command
Rome, New York**

DISCLAIMER NOTICE

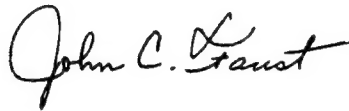


THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

RL-TR- 95-295, Vol.II (of four), has been reviewed and is approved for publication.

APPROVED:



JOHN C. FAUST
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify Rome Laboratory/ (C3AB), Rome NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE April 1996		3. REPORT TYPE AND DATES COVERED Final Aug 90 - Jun 94	
4. TITLE AND SUBTITLE ROMULUS, A COMPUTER SECURITY PROPERTIES MODEL ENVIRONMENT: Romulus Theories				5. FUNDING NUMBERS C - F30602-90-C-0092 PE - 35167G PR - 1065 TA - 01 WU - 03	
6. AUTHOR(S) S. Brackin, S. Foley, L. Gong, B. Hartman, A. Heff, G. Hird, D. Long, D. McCullough, I. Meisels, D. Rosenthal, I. Sutherland, and A. Weitzman					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Odyssey Research Associates, Inc. 301 Dates Drive Ithaca NY 14850-1326				8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3AB 525 Brooks Rd Rome NY 13441-4505				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-95-295, Vol II (of four)	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: John C. Faust/C3AB/(315) 330-3241					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The Romulus security properties modeling environment contains tools, theories, and models that support the high-level design and analysis of secure systems. The Romulus nondisclosure tool supports development and analysis of distributed composite security models and their properties. The Romulus modeling approach establishes the models on a solid theoretical basis and uses formal mathematical tools to aid in the analysis. Romulus allows a user to express a model of a secure system using a formal specification notation that combines graphics and text. Verification of the model proves that it satisfies its critical properties. The user verifies the model by using a combination of automatic decision procedures and interactive theorem proving. The primary emphasis in the current system is the analysis of multilevel trusted system models to see if they satisfy nondisclosure properties. Romulus also includes a tool for formally specifying and verifying authentication protocols. This tool can be used to reason about the beliefs of the parties engaged in a protocol in order to analyze whether the protocol achieves the desired behavior. The Romulus theories include formal theories of nondisclosure, integrity, and (see reverse)					
14. SUBJECT TERMS Computer security, Nondisclosure, Integrity, Availability, Security properties modeling, Information flow analysis, Design verification, Authentication protocol analysis, (see reverse)				15. NUMBER OF PAGES 160	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL		

13. (Cont'd)

availability security. The Romulus library of models demonstrates the application of these theories.

14. (Cont'd)

Multilevel security, Security policy

Preface

This four volume report describes Romulus, a security modeling environment. Romulus includes a tool for constructing graphical hierarchical process representations; an information flow analyzer; a process specification language; and techniques to aid in doing proofs of security properties. Romulus also contains tools for the specification and analysis of authentication protocols. Using Romulus, a user can develop and analyze security models and properties. The foundations of Romulus are formal theories of security; applications of these theories are demonstrated in a library of models.

In this volume, we assume that the reader has some familiarity with the Romulus tools, the HOL system, and security issues in general.

Organization of the Romulus Documentation Set

This volume is Volume II of a four volume documentation set; this volume describes the Romulus theories for nondisclosure, integrity and availability. Volume I contains an overview of the Romulus environment. Volume III describes the Romulus library of models. Volume IV is the Romulus User's Manual; it contains descriptions of the Romulus tools, how to use them, and tutorial examples.

Organization of This Volume

In this volume, we describe the formal theories of nondisclosure, integrity, and availability that form the basis for the Romulus tools and models. The Romulus theories of nondisclosure are described in Chapter 2, the Romulus theories of integrity are described in Chapter 3, and the Romulus theories of availability are described in Chapter 4.

Conventions

This document set uses the following conventions. Computer code, specifications, program names, file names, and similar material are typeset using a typewriter font. Interactive computer sessions are surrounded by a rounded box. Within this box, user input is typeset using an *italic typewriter* font; computer output is typeset using the typewriter font. Some computer

output has been reformatted for presentation purposes; it may not appear in this document exactly as it appears on your screen.

Acknowledgements

Portions of this volume were extracted from previous ORA reports and papers, specifically, [41], [40], [7], [42], [17], [58] and [52].

Contents

1	Introduction	1
2	Theories of Nondisclosure	2
2.1	Restrictiveness	2
2.1.1	Introduction	3
2.1.2	Introducing Restrictiveness	10
2.1.3	Restrictive State Machines	13
2.1.4	Hooking Up Machines	14
2.1.5	Conclusions	16
2.2	Shared-State Restrictiveness	17
2.2.1	Overview	18
2.2.2	Shared-state Machines	18
2.2.3	Shared-State Restrictiveness	19
2.2.4	Shared-State Hookup	20
2.2.5	Composability	22
2.2.6	Fundamental Security Theorem	23
2.2.7	Afterword	25
2.3	Server/Client Restrictiveness	27
2.3.1	Introduction	27
2.3.2	Theory	29
2.3.3	Practice	48
2.3.4	Summary	52
2.4	Implementation	52
2.4.1	Background	52
2.4.2	Server-Process Restrictiveness	53

3	Theories of Integrity	70
3.1	Integrity Models	70
3.1.1	Requirements of Integrity Theories	71
3.1.2	Techniques for Ensuring Integrity	72
3.1.3	System Representations for Integrity	79
3.1.4	Formal Integrity Properties	81
3.1.5	Examples	92
3.2	Authentication Protocols	98
3.2.1	Authentication	99
3.2.2	Protocol Descriptions	99
3.2.3	Protocol Specifications	101
3.2.4	The <code>crypto_90</code> Theory	104
3.2.5	Analyzing a Protocol	115
4	Theories of Availability	118
4.1	Requirements of Availability Theories	119
4.2	Techniques for Ensuring Availability	120
4.2.1	Replication	120
4.2.2	Static and Dynamic Resource Allocation	120
4.3	System Representations for Availability	121
4.3.1	State Machine Representation	121
4.3.2	Representing Timing Properties	121
4.4	Formal Availability Properties	123
4.4.1	t -Fault Tolerance	123
4.4.2	Timeliness Properties	126
4.4.3	Dynamic Security Properties	128
4.5	Example	132
4.5.1	Fault Tolerant Sensor	132
A	Probability	140
	Bibliography	142

List of Figures

2.1	Deducibility Security is not Composable	9
2.2	PSL Process Semantics	59

Chapter 1

Introduction

The methods and models that Romulus uses are based, where possible, on a formal mathematical foundation. This volume describes the security theories that form the basis of the Romulus environment. These theories are grouped into three general areas, theories of nondisclosure, theories of integrity, and theories of availability (also known as service assurance or denial of service). Each of these areas covers a broad spectrum of issues and none of them can be covered by a single theory. The Romulus approach uses multiple theories, each focusing on specific aspects of nondisclosure, integrity or availability. This volume describes those formal theories that have been developed as part of the Romulus project. The theories presented here do not attempt to provide a comprehensive answer to questions of nondisclosure, integrity, or availability. Other theories, covering other aspects of nondisclosure, integrity, or availability could well be included in future versions of Romulus. Applications of the theories described in this volume can be found in Volume III, the Romulus library of models.

This volume is divided into three main chapters, one each for the theories of nondisclosure, integrity and availability. Chapter 2 starts with a discussion of the primary Romulus nondisclosure theory, continues with two variations on this theory and concludes with a discussion of the current implementation of the Romulus the primary nondisclosure theory. Chapter 3 discusses authentication protocols in addition to the Romulus theories of integrity. Chapter 4 discusses the Romulus theories of availability.

Chapter 2

Theories of Nondisclosure

In this chapter we present the formal theories of nondisclosure security used in Romulus, which are based on *restrictiveness*. Restrictiveness is a *hookup security property*, which means that a collection of secure restrictive systems when “hooked together” form a secure restrictive composite system. We believe that the security enforcement and composability provided by restrictiveness make it an attractive choice for a security policy on trusted systems and processes.

In section 2.1 we give the definition of restrictiveness based on a state machine model. This section describes why restrictiveness is used as the basis for our formal theories, gives a formal definition of restrictiveness, and proves that restrictive processes are composable. In section 2.2 and section 2.3 we describe variations on this approach that handle shared resources. The first variation, shared state restrictiveness, uses a different process model than restrictiveness. The second variation, server/client restrictiveness, uses the same process model as restrictiveness, but decomposes the security property to handle servers and clients. section 2.4 discusses the current Romulus implementation of restrictiveness.

2.1 Restrictiveness

In this section, we discuss the background for the Romulus theory of security and sketch a proof of the main result, that restrictiveness is a composable security property [23], [24], [25]. This material previously appeared in [36]

and [25].

2.1.1 Introduction

Multilevel security requires that sensitive information be disclosed only to authorized personnel. In the “paper world”, this is enforced by assigning to each document and each employee a *security level* indicating sensitivity and authority. Commonly used levels in the government are *unclassified*, *confidential*, *secret* and *top_secret*. The levels form a partially ordered set, so that an employee can be said to be authorized to read a document only if his level is greater than or equal to that of the document.

Multilevel security becomes more complicated for information processing systems because not all information is in the form of documents and not all consumers of information are employees. Generally for such systems the problem of multilevel security consists of two aspects:

1. *Access control*—determining who can see information of a given sensitivity leaving the system
2. *Correct labeling*—determining the sensitivity of information entering and leaving the system

The second issue, correct labeling, becomes much more important in computer systems because of the possible presence of *Trojan Horse* programs. A Trojan Horse program is a malicious program that when run by a high level user will try to surreptitiously obtain classified information from that user and convey it to an accomplice, usually the user who programmed the Trojan Horse. These programs often masquerade as useful programs such as word processors or compilers.

Rather than inspect each program on a system to see if it is a Trojan Horse program (it may not be possible to decide by inspection whether a program is a Trojan Horse or not), people instead try to build computer systems which are secure even in the presence of malicious programs.

In this section we describe a security property that addresses these issues, called *restrictiveness*. It is *composable*, which means that for a collection of trusted processes “hooked up” to make a system, or for a collection of systems “hooked up” to make a network, the system or network is secure if each component is.

Using restrictiveness as a definition of security for trusted systems provides confidence in building large, complex systems from smaller, easier to verify trusted components. Security is modularized and so becomes more manageable. We first consider two earlier models for security, and point out some of their short-comings.

2.1.1.1 The Bell-LaPadula Model: Trusted and Untrusted Processes

The issues of access control and correct labeling are addressed by the Bell-LaPadula model [6]. In this model, all entities involved with a computer system—users, files, programs, etc.—are divided into two classifications: *subjects* and *objects*. Subjects are the active entities, such as users and processes which are capable of reading and modifying system state information, while objects are passive containers for information, such as files. Subjects which are processes are further divided into *trusted processes* and *untrusted processes*.

Rules for Untrusted Subjects To enforce security for untrusted processes, the Bell-LaPadula model requires that every object and every untrusted process be assigned a security level. The model then demands that

1. An untrusted process may only read from objects of lower or equal security level. (Motto: Read down)
2. An untrusted process may only write to (or modify) objects of greater or equal security level. (Motto: Write up)

The *read down* rule insures that a process is only allowed to read information it is entitled (by its security classification) to read. The *write up* rule insures that all objects are correctly labeled; it is impossible to put information into an object which comes from a source whose level is higher than the security label on the object.

In the Bell-LaPadula model, users are interpreted as untrusted subjects, except that allowance is made for users to act at any security level less than their maximum. For example, a *secret* user may login as a *secret* or *unclassified* subject, but not as a *top-secret* subject.

Trusted Processes The answers the Bell-LaPadula model provides are incomplete since it does not provide guidance for determining the sensitivity of information coming from the *trusted processes* of a system. A trusted process is any process which is exempted from the stringent requirements enforced on untrusted processes. Since trusted processes are not bound by the normal rules, it is necessary to have reason to trust that they do not behave maliciously.

Trusted processes are needed whenever an activity potentially involves information at several different security levels. For example, the file server must be able to read and write files at all different levels, and so cannot possibly be bound by the access control rules given above. In the Bell-LaPadula model, the need for such multi-level processes was recognized, but no standard security rules for the behavior of such processes were given.

To fill this gap, it is desirable to have a security property for a trusted process that guarantees that information leaving the process is correctly labeled: that high-level information does not “leak” into low-level outputs. We next consider a candidate for such a property.

2.1.1.2 Noninterference and Deducibility Security

A *multilevel process* is a system which takes in information of different security levels, processes it and outputs information of different security levels. Note that this description can equally well describe a trusted process, or an entire computer system, or a network of computer systems. A general framework for defining security for such systems, called *deducibility security* is found in Sutherland [56]. In this section, we will present an informal description of a special case of Sutherland’s model.

We will assume that all the sensitive information contained in a process enters the process in the form of discrete *inputs* labeled with a security level indicating their sensitivity, and that information leaves the process in the form of labeled *outputs*.¹ The word *event* will be used to refer to an input or an output.

For each security level l we will call the events of level less than or equal to l the *view* for that level, and all other events we will say are *hidden* from that level. By the assumption that access control is enforced on the untrusted

¹We are ignoring the possibility that new sensitive information might be created *inside* the process.

parts of the system, we can know that users of level l are unable to see any events outside their own view. The hidden events for a level l are the inputs made by users of higher (or incomparable) level. Under these circumstances, we will say that a multilevel process is *deducibility secure* if for each security level l and for every sequence of events possible in some history of the process

*The information contained in the events in the view for level l
does not reveal anything about information in inputs hidden from
level l .*

Goguen-Meseguer Noninterference To formalize this statement, it is necessary to say what it means for one set of observations (the sequence of events in the view of some level) not to reveal anything about some other source of information (the sequence of inputs hidden from that level). Goguen and Meseguer [15] formalized this notion by saying that the hidden high-level inputs cannot *interfere* with the low-level view, the sequence of low-level outputs. For their model, they considered systems such that the sequence of outputs produced by the system is a deterministic function of the input sequence. A system is then said to obey the Goguen-Meseguer noninterference policy if, for any possible input sequence, if one removes all the high-level inputs, the low-level part of the corresponding output sequence will be unchanged.

Deducibility Security Goguen-Meseguer noninterference is not as general as one would like, since it is only meaningful for deterministic systems (ones where what is observed is completely determined by the inputs). A more general definition is the requirement of *deducibility security* [56]. With the choices of the views and the hidden information given above, we say that a system is deducibility secure if any possible set of observations in the view is *consistent* with any possible sequence of hidden inputs. That is, it is impossible for a user to “rule out” any sequence of hidden inputs. Since we intend to consider cases, such as concurrency, where there is nondeterminism, we will base our security theory on deducibility security rather than noninterference.

Deducibility security prevents leaks due to Trojan Horses. If an entire system, both verified trusted and untrusted software (including any Trojan Horses that may be lurking around) is deducibility secure, and it

initially has no classified information in it, then no unauthorized user of that system will ever learn any classified information through the system. In other words, any system which allows illegal information flow through the system is not deducibility secure. The informal argument goes as follows:

Suppose that a low level user learns through the system some piece of high level information on the system. This means that the user's view of the system behavior, call it b , has revealed that some condition C holds for the system, and the fact that this condition holds is classified information. Since we are only considering systems which do not create any new classified information internally, if C holds, then there must have been an earlier moment at which high level information was put into the system by high level users; some action must have been taken by the high level user which caused condition C to hold.

Therefore, behavior b allows the low level user to infer C , which allows him to infer that high level users performed some action. So the high level behavior described by "doing nothing" can be ruled out. Since some high level behavior can be ruled out, the system is not deducibility secure.

If deducibility security prevents all leaks due to Trojan Horses, then what more could one want in a definition for multilevel security? Well, in practical terms, one does not verify everything, one only verifies certain "security relevant" portions of the system. Is it possible that a user can combine information obtained from two different components of a system in order to learn information that he couldn't receive from either system in isolation? The answer turns out to be yes, as the next section shows.

Deducibility Security Is Not Preserved by Hookup A multilevel system may have many interacting trusted processes, so it is not sufficient to guarantee that each process is secure in isolation; it is necessary to also show that several processes working together cannot conspire to violate security. In other words, for trusted processes, it is necessary to have a definition of security that is *composable*. While deducibility security may be a good overall definition of security for an entire system, it unfortunately is not composable. In this section we will demonstrate this result by showing two processes which

alone are deducibility secure, but which when “hooked up” form a composite system which is not secure.

In the following example, we will consider systems with only two levels: *high* and *low*.

Conventions and Notations for Systems: To illustrate the possible behavior of systems, let us introduce a pictorial notation for the traces, or possible histories, of systems. We depict a trace of a system by giving a timeline running vertically, with the future of the system toward the top and with the past of the system toward the bottom. Horizontal vectors directed toward or away from the time line of a system represent inputs to and outputs from that system, respectively. We will use unbroken lines to represent *low* inputs and outputs, and broken lines to represent *high* inputs and outputs. To represent two systems operating in parallel, we show their timelines together. Messages sent from one machine to another will be shown as a horizontal arrow pointing away from the time line of the sending system and toward the time line of the receiving system.

A Counterexample: Consider a system, called \mathcal{A} , which has the following set of traces: each trace starts with some number of high-level inputs or outputs followed by the low-level output *stop* followed by the low-level output *odd* (if there have been an odd number of high-level events prior to *stop*) or *even* (if there have been an even number of high-level events prior to *stop*). The high-level outputs and the output of *stop* leave via the right channel of the process, and the events *odd* and *even* leave via the left channel. The high-level outputs and the output of *stop* can happen at any time.

Two possible event sequence of system \mathcal{A} is portrayed in the top left corner of Figure 2.1.

System \mathcal{A} actually is deducibility secure; regardless of high-level inputs, the possible low-level sequences are (1) *stop* followed by *odd* or (2) *stop* followed by *even*. A high-level input does not affect these possibilities, because it is always possible for such an input to be followed by a high-level output, and the pair would leave the low-level outputs unaffected.

System \mathcal{B} behaves exactly like system \mathcal{A} , except that:

- its high-level outputs are out its left channel
- its *even* and *odd* outputs are out its right channel.
- *stop* is an input to its left channel, rather than an output.

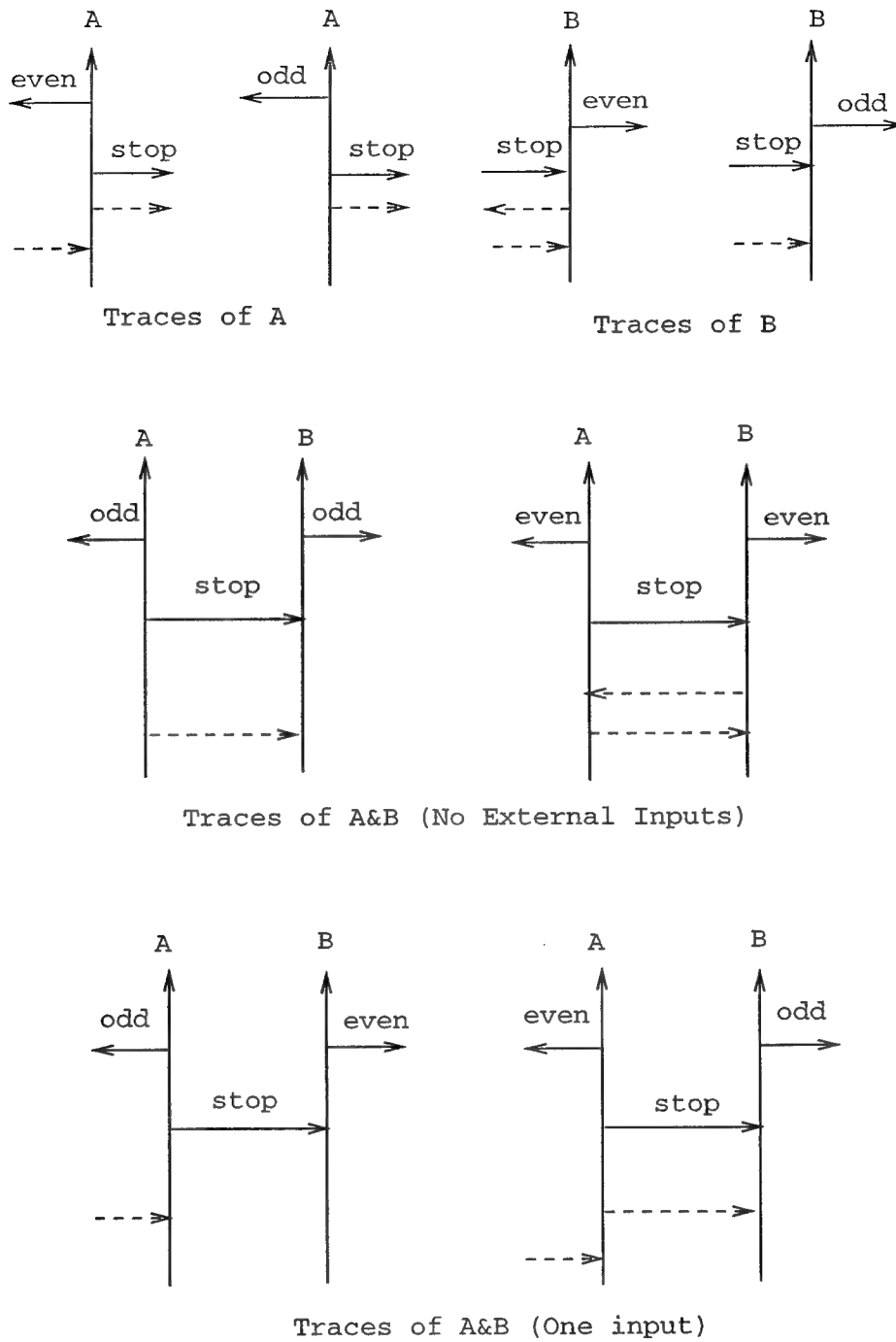


Figure 2.1: Deducibility Security is not Composable

System \mathcal{B} , like system \mathcal{A} , is deducibility secure. A typical event sequence of system \mathcal{B} is shown in the upper right corner of Figure 2.1.

If systems \mathcal{A} and \mathcal{B} are connected, so that the left channel of \mathcal{B} is connected to the right channel of \mathcal{A} then we have the situation pictured in the bottom of Figure 2.1.

Now we see that the combined system is no longer deducibility secure: Since the number of shared high-level signals is the same for \mathcal{A} and \mathcal{B} , the fact that \mathcal{A} says *odd* while \mathcal{B} says *even* (or vice-verse) means that there has been at least one high-level input from outside. If all high-level inputs are deleted, then systems \mathcal{A} and \mathcal{B} will necessarily both say *even* or both say *odd*. By looking at the low-level events, a user can deduce something about the high-level inputs.

Composition allows one to turn small information leaks into large leaks. Although the example above leaks only a small amount of information, it is not difficult to devise schemes for combining several slightly leaky systems to get a very leaky system. For some examples of how insecurities can multiply see [22].

2.1.2 Introducing Restrictiveness

An important thing to notice about the failure of composability for deducibility security is that, although a system obeying the property insures that no *single* high-level input will affect the future low-level behavior, it does not guarantee that a *pair*, consisting of a high-level input followed immediately by a low-level input, will have the same effect on the low-level behavior as the low-level input alone. From Figure 2.1, it is clear that system \mathcal{B} does not insure this latter, stronger form of noninterference. For example, the pair consisting of a high-level input followed by *stop* does *not* have the same effect as *stop* alone.

The additional requirement can be intuitively understood as follows: Only some facts about the past of a system are relevant for the future low-level behavior of the system. These relevant facts can be thought of as defining the “low-level state” of the system. The requirement of noninterference is that a high-level input may not change the low-level state of the system. Therefore, the system should respond the same to a low-level input whether or not a high-level input was made immediately before.

Restrictiveness is a property of systems which formalizes this requirement

². In the next sections, we formalize restrictiveness as a property of state machines³, and prove that it is composable.

2.1.2.1 State Machines

A *state machine* is a way of describing a computer system in terms of its internal structure, and its input-output behavior. To describe a system as a state machine, one must give

1. The set of possible *states*.
2. The set of possible *events*, the inputs, outputs and internal signals of the system.
3. The set of possible *transitions*.
4. The *initial state*.

A transition is denoted by

$$\sigma_0 \xrightarrow{e} \sigma_1$$

where σ_0 is the state of the machine before the transition, e is the accompanying event for the transition, and σ_1 is the state of the machine after the transition. A sequence of transitions starting in σ_0 and ending in σ_n , involving events $[e_1, \dots, e_n]$ is denoted by $\sigma_0 \xrightarrow{[e_1, \dots, e_n]} \sigma_n$. We say that σ_0 *can accept* event e if for some state σ_1 , $\sigma_0 \xrightarrow{e} \sigma_1$.

The Traces: Traces of a state machine are all sequences of events γ such that for some state σ_1 $start \xrightarrow{\gamma} \sigma_1$, where *start* is the initial state.

²Goguen and Meseguer's notion of noninterference formalized this notion for deterministic state machines. For the class of machines they considered, restrictiveness and noninterference in their sense agree.

³In [21] a similar property was defined solely in terms of the traces of a system; states were not involved.

Input Total State Machines A state machine is said to be *input total* if in any state it can accept an input. The significance of this property for our purposes is that for an input total machine, one can only learn about its state by watching its outputs; no information is conveyed to the user by accepting inputs. In contrast, in systems which are not input total, one learns something about the state of the system whenever an input is accepted; namely that it is in an accepting state for that input.

By restricting our attention to input total processes, we achieve a technical simplification for our theory of security: information enters a system through its inputs, and leaves a system through its outputs. We will make input totality a condition for a state machine to be restrictive, but this is not intended to imply that only such machines are secure. Rather, restrictiveness as a definition of security only applies to input total machines.

2.1.2.2 Security for State Machines

Once again, we will only consider the case of two security levels, *low* and *high*.⁴ To prevent a low-level user from discovering information he is not allowed to know, we first need to specify the high-level *state information*, and *event information*. We can summarize this information through the use of two equivalence relations, one on states and one on event sequences.

If σ_1 and σ_2 are two states, then we say $\sigma_1 \approx \sigma_2$ if the states differ only in their high-level information. In other words, if to each state variable we assign either the security level *high* or *low*, then $\sigma_1 \approx \sigma_2$ if the values of all low-level variables are the same in the two states.

If γ_1 and γ_2 are two sequences of events, then we say that $\gamma_1 \approx \gamma_2$ if the two sequences agree for low-level events. For example, if the event a is low, and the event b is high, then the following three events are all considered equivalent:

$$[a, b, b, a] \approx [b, a, b, a] \approx [a, a]$$

A low-level user's view of the system, the state information he may know, and the events he may see, is completely determined by the equivalence relations on states and event sequences (both of which we will denote by \approx).

⁴State machines with multiple security levels are known as *rated* state machines.

2.1.3 Restrictive State Machines

A state machine is defined to be *restrictive* for the view determined by \approx if

1. It is input total.
2. For any states σ_1, σ'_1 and σ_2 , and for any two input sequences β_1 and β_2 , if

$$(a) \sigma_1 \xrightarrow{\beta_1} \sigma'_1$$

$$(b) \sigma_2 \approx \sigma_1$$

$$(c) \beta_1 \approx \beta_2$$

then for some state σ'_2

$$(a) \sigma_2 \xrightarrow{\beta_2} \sigma'_2$$

$$(b) \sigma'_2 \approx \sigma'_1$$

3. For any states σ_1, σ'_1 and σ_2 , and for any output sequences γ_1 , if

$$(a) \sigma_1 \xrightarrow{\gamma_1} \sigma'_1$$

$$(b) \sigma_2 \approx \sigma_1$$

then for some state σ'_2 and some output sequence γ_2

$$(a) \sigma_2 \xrightarrow{\gamma_2} \sigma'_2$$

$$(b) \sigma'_2 \approx \sigma'_1$$

$$(c) \gamma_2 \approx \gamma_1$$

Rules 2 and 3 say, roughly, that “Equivalent states are affected equivalently by equivalent inputs, produce equivalent outputs and then remain equivalent”. This is a noninterference assertion; that high-level inputs and high-level state information cannot affect the behavior of the system, as viewed by a low-level user. Restrictiveness thus generalizes the Goguen-Meseguer definition of noninterference to nondeterministic systems. (For the particular kind of deterministic machines that Goguen and Meseguer considered, restrictiveness and noninterference reduce to the same property.)

One should note that an immediate consequence of 2 is that if β_1 is a high-level input sequence, then it must not affect the state at all. Also, in rule 3, it is easy to show by induction that it is enough to consider cases in which γ_1 (but not necessarily γ_2) consists of a single event.

2.1.4 Hooking Up Machines

If \mathcal{A} and \mathcal{B} are two machines, then hook them up by sending some of the outputs of \mathcal{A} to be inputs to \mathcal{B} , and vice-versa. These common communication events are internal events of the composite machine, which will be treated the same as output events for the purposes of security. The inputs of the composite machine are the inputs of either component machine which are not supplied by the other. The states of the combined machines are pairs $\langle \sigma, \nu \rangle$, where σ is a state of \mathcal{A} , and ν is a state of \mathcal{B} .

The Events: An event of the composite machine is any event from either component machine. For any sequence of events γ from the composite machine, we will let $\gamma \upharpoonright E_{\mathcal{A}}$ be the sequence of events in γ engaged in by machine \mathcal{A} , and $\gamma \upharpoonright E_{\mathcal{B}}$ be the sequence of events engaged in by \mathcal{B} . Because of the shared communication events, some events from gamma will occur in $\gamma \upharpoonright E_{\mathcal{A}}$ and in $\gamma \upharpoonright E_{\mathcal{B}}$.

The Transitions: $\langle \sigma, \nu \rangle \xrightarrow{\gamma} \langle \sigma', \nu' \rangle$ is a valid transition for the composite machine if and only if $\sigma \xrightarrow{\gamma \upharpoonright E_{\mathcal{A}}} \sigma'$ and $\nu \xrightarrow{\gamma \upharpoonright E_{\mathcal{B}}} \nu'$ are valid transitions of the component machine. This notion of hookup, or parallel composition, is taken from CSP [19]. It assumes that the only correlation between state transitions of the two components is through the shared communication events.

Security: The equivalence relations for the composite machine are inherited from those of the component machines:

- $\langle \sigma, \nu \rangle \approx \langle \sigma', \nu' \rangle$ if and only if $\nu \approx \nu'$ and $\sigma \approx \sigma'$.
- $\gamma \approx \gamma'$ if and only if $\gamma \upharpoonright E_{\mathcal{A}} \approx \gamma' \upharpoonright E_{\mathcal{A}}$ and $\gamma \upharpoonright E_{\mathcal{B}} \approx \gamma' \upharpoonright E_{\mathcal{B}}$.

2.1.4.1 Restrictiveness is Composable

If state machines \mathcal{A} and \mathcal{B} are restrictive, then a composite machine formed from hooking them up is restrictive.

The composite machine is input total. If β is any sequence of inputs for the composite machine, then $\beta \upharpoonright E_A$ is a sequence of inputs for \mathcal{A} , and $\beta \upharpoonright E_B$ is a sequence of inputs for \mathcal{B} . If $\langle \sigma, \nu \rangle$ is any starting state, then $\beta \upharpoonright E_A$ is a sequence of inputs for \mathcal{A} , and $\beta \upharpoonright E_B$ is a sequence of inputs for \mathcal{B} . Since \mathcal{A} and \mathcal{B} are input total, there are states σ' and ν' such that $\sigma \xrightarrow{\beta \upharpoonright E_A} \sigma'$ and $\nu \xrightarrow{\beta \upharpoonright E_B} \nu'$. Therefore, $\langle \sigma, \nu \rangle \xrightarrow{\beta} \langle \sigma', \nu' \rangle$.

Inputs affect equivalent states equivalently. If $\langle \sigma_1, \nu_1 \rangle$, $\langle \sigma'_1, \nu'_1 \rangle$, and $\langle \sigma_2, \nu_2 \rangle$ are states and β_1 and β_2 are input sequences, then since \mathcal{A} and \mathcal{B} are restrictive, there must be states σ'_2 and ν'_2 such that

1. (a) $\sigma_2 \xrightarrow{\beta_2 \upharpoonright E_A} \sigma'_2$
 (b) $\nu_2 \xrightarrow{\beta_2 \upharpoonright E_B} \nu'_2$
2. (a) $\sigma'_2 \approx \sigma'_1$
 (b) $\nu'_2 \approx \nu'_1$

Therefore,

1. $\langle \sigma_2, \nu_2 \rangle \xrightarrow{\beta_2} \langle \sigma'_2, \nu'_2 \rangle$
2. $\langle \sigma'_2, \nu'_2 \rangle \approx \langle \sigma'_1, \nu'_1 \rangle$

Equivalent states produce equivalent outputs, which lead again to equivalent states. As remarked earlier, it is sufficient to consider outputs of single events. Suppose then that e is an output, and that

1. $\langle \sigma_1, \nu_1 \rangle \xrightarrow{e} \langle \sigma'_1, \nu'_1 \rangle$
2. $\langle \sigma_1, \nu_1 \rangle \approx \langle \sigma_2, \nu_2 \rangle$

An output for the composite machine must be an output for one of the component machines. We assume then that e is an output from \mathcal{A} ; the other case is handled similarly.

Since \mathcal{A} is restrictive, and $\sigma_1 \xrightarrow{e} \sigma'_1$, and $\sigma_2 \approx \sigma_1$, then for some state σ'_2 and some output sequence γ_A : $\sigma_2 \xrightarrow{\gamma_A} \sigma'_2$, and $\sigma'_2 \approx \sigma'_1$, and $\gamma_A \approx [e]$.

Now, since the sequence γ_A is an output sequence, any events shared by both \mathcal{A} and \mathcal{B} must be inputs to \mathcal{B} . Since $\gamma_A \approx [e]$, it follows that $\gamma_A \upharpoonright E_B \approx [e] \upharpoonright E_B$. Therefore, we have

1. $\nu_1 \approx \nu_2$
2. $\nu_1 \xrightarrow{[e] \uparrow E_B} \nu'_1$
3. $\gamma_A \uparrow E_B \approx [e] \uparrow E_B$

From the fact that \mathcal{B} is restrictive, we get that for some state ν'_2

1. $\nu_2 \xrightarrow{\gamma_A \uparrow E_B} \nu'_2$
2. $\nu'_2 \approx \nu'_1$

Therefore, we have that

1. $\langle \sigma_2, \nu_2 \rangle \xrightarrow{\gamma} \langle \sigma'_2, \nu'_2 \rangle$
2. $\langle \sigma'_2, \nu'_2 \rangle \approx \langle \sigma'_1, \nu'_1 \rangle$

2.1.5 Conclusions

In analogy with the Bell-LaPadula model, we can require that every untrusted process be assigned a security level, and also require that every output be greater than or equal to this level (motto: send up), and that every input be less than or equal to this level (motto: receive down). It is easy to see that every such untrusted process is *manifestly secure*; it is necessarily restrictive, if we assume that the level of all state information is the same as the level of the process. Therefore, an immediate consequence of the hookup theorem for restrictive machines is that:

If every component of a system is proved restrictive, or is untrusted and manifestly secure, then the entire system is restrictive. Therefore, any Trojan Horse in a restrictive system is harmless, as long as it is only allowed to send up and receive down.

In this section we have argued for the need for a formal definition of security that is applicable for a wide range of processes, systems, and networks. The Bell-LaPadula model is not sufficient for this purpose, because while it defines security in terms of access controls, it does not provide sufficient guidance for the security of trusted processes. Sutherland's deducibility security

provides a general definition of security for total systems, but unfortunately is not composable, and so cannot be applied to components of a large system. The final property described, restrictiveness, is generally composable and so can be used as a definition of security for small processes and entire systems.

2.2 Shared-State Restrictiveness

Restrictiveness is a security property that can be used when the process to be modeled can be modeled as a state machine. There are situations, however, where the use of restrictiveness is difficult or cumbersome because it is difficult or cumbersome to model with process with a state machine. The modeling of shared resources is such a case. This section introduces a version of restrictiveness, called shared-state restrictiveness, based on machines that share state information. Like restrictiveness, shared-state restrictiveness is a composable property, so that the result of properly connecting shared-state restrictive components is shared-state restrictive. Unlike restrictiveness, shared-state restrictiveness describes process behavior purely in terms of transformations on system variables rather than in terms of input and output events.

Shared-state restrictiveness distinguishes the system variables that can be changed by entities outside the system (*labeled* variables) from the system variables that can be changed only by the system itself (*unlabeled* variables). Input events in ordinary restrictiveness are analogous to changes in labeled variables made by entities outside the system; output events in restrictiveness are analogous to changes in labeled variables made by the system itself. Shared-state restrictiveness can hold, though, for systems in which “input” changes to system variables and “output” changes to system variables can occur simultaneously, and in which more than one system variable can change at a time.

Shared-state restrictiveness defines all sensitivity-level information for system variables in terms of a *security structure* projection function that captures only the information in system variables that is legitimately accessible to one having clearance at a particular level. This function subsumes both ordinary restrictiveness’ functions assigning security levels to input and output events and its projection functions capturing just the information in system variables needed to give the future system behavior visible at a

particular level.

Shared-state restrictiveness is a natural tool for analyzing systems involving shared state. The simplest way of applying ordinary restrictiveness to systems involving shared state is to introduce new processes that manage the shared resources; this approach raises issues such as whether these new processes are always guaranteed to reply to requests and whether these new processes are themselves secure. section 2.3 describes this approach to shared resources.

2.2.1 Overview

In section 2.2.2 we describe the formalism we will use for representing state machines and their security parameters. In section 2.2.3, we give the definition of shared-state restrictiveness. In section 2.2.4, we define what it means to hook up an ensemble of state machines by sharing state. In section 2.2.5, we prove that the secure hookup of a collection of shared-state restrictive machines is shared-state restrictive. In section 2.2.6, we prove that shared-state restrictiveness implies an information security property.⁵

For this section, we will assume that there are only two security levels, “high” and “low”. We will describe how to generalize to an arbitrary partially ordered set of levels in section 2.2.7.

2.2.2 Shared-state Machines

A *shared-state machine* consists of:

- a set of *state variables*, each of which has an associated nonempty set of values called the variable’s *type*; an assignment of the set of state variables to values that assigns each variable to an element of its type will be called a *state* of the machine

⁵We include this proof because shared-state restrictiveness is not *prima facie* an information security property. It’s clear from the definition that it has *something* to do with information security, but it’s not intuitively clear that it ensures nondisclosure. The result in section 2.2.6 is meant to demonstrate that restrictiveness ensures the kind of nondisclosure that is desired.

- a subset of the state variables called the *labeled variables*; these are the variables that can be shared with other machines; variables that are not labeled variables will be called *unlabeled variables*
- a nonempty set of *initial states*, interpreted as the set of all states the machine can start in
- a binary relation on states called the *state transition relation*; this defines what states a machine can go to from a given state; the pairs in the state transition relation will often be referred to as the *legal transitions*
- a subset of the state transition relation called the *inner state transitions*; these are the state transitions that are thought of as being initiated by the shared-state machine, rather than by the shared-state machine's environment; all state transitions that are not inner will be called *outer*

The *security parameters* of a shared-state machine consist of the following: for each labeled variable, an equivalence relation on the type of that variable. This equivalence relation is meant to be a general mechanism for "separating out" the "low data" contained in the value of a labeled variable. Two values in the type of a labeled variable being equivalent means that the values contain the same low data.

2.2.3 Shared-State Restrictiveness

First, we will define a few auxiliary terms.

We define the *reachable states* of a shared-state machine in the usual way, that is, a state s is reachable if and only if there exists a nonempty sequence of states such that (1) the first state is an initial state, (2) every consecutive pair of states is a legal transition, and (3) the last state is s .

We will use the term *unlabeled state* to refer to an assignment of each unlabeled variable of a state machine to an element of its type. The unlabeled state induced by a given state will be called the *unlabeled part* of the state.

If \simeq_v is the equivalence relation on the labeled variable v , and \simeq is an equivalence relation on the unlabeled states of the shared-state machine, then the *induced equivalence relation* on states is the equivalence relation that makes two states equivalent if (1) for each labeled variable v , the values assigned to v by the two states are equivalent by \simeq_v , and (2) the unlabeled parts of the two states are equivalent by \simeq .

If \equiv is any equivalence relation on states, and s_1 and s_2 are states with $s_1 \equiv s_2$, and $\langle s_1, t_1 \rangle$ is a legal transition of the shared-state machine, we say that the transition *shifts to s_2* or *can be shifted to s_2* if there exists a sequence of states $\langle t^{(1)}, \dots, t^{(n)} \rangle$ such that

- $t^{(1)} = s_2$
- every consecutive pair of states in the sequence is a legal transition
- $t^{(n)} \equiv t_1$

We call the sequence $\langle t^{(1)}, \dots, t^{(n)} \rangle$ a *shifting* of $\langle s_1, t_1 \rangle$.

Now, we will give the definition of shared-state restrictiveness.

We say that a shared-state machine (with security parameters) is *shared-state restrictive* if and only if there exists an equivalence relation \simeq on the unlabeled states of the machine such that if \equiv is the induced equivalence relation on states, then for every pair of reachable states s_1 and s_2 with $s_1 \equiv s_2$, every inner transition $\langle s_1, t_1 \rangle$, can be shifted to s_2 with a shifting consisting entirely of inner transitions.

2.2.4 Shared-State Hookup

In this section we define what it means to hook up an ensemble of state machines by sharing state. To simplify our exposition, we will assume that there is a universal set of state variables with associated types, and that the state variables of all shared-state machines are some subset of this universal set, with the same types associated. We will start with a few auxiliary definitions.

Given a collection of state machines, let V be the union of the state variables of the machines in the collection. We define a *common state* of

the collection to be an assignment of each element of V to an element of its associated type. If s is a common state and M is a machine in the collection, we denote the restriction of s to the state variables of M by $s[M]$.

We define the *common initial states* to be the set of all common states s such that for all machines M in the collection, $s[M]$ is an initial state of M . We define the *common transitions* to be the set of all pairs of common states $\langle s, t \rangle$ such that for each machine M in the collection, $\langle s[M], t[M] \rangle$ is a legal transition of M . We define the *common reachable states* to be the set of all states reachable from a common initial state by a sequence of common transitions. It is immediate that if s is a common reachable state and M is a machine in the collection, $s[M]$ is a reachable state of M .

We say that a collection of shared-state machines is *compatible* if and only if:

- The set of common initial states of the collection is nonempty.
- For every pair of distinct machines M_1 and M_2 in the collection:
 - any variable shared by M_1 and M_2 is a labeled variable of both of them, and
 - for any common reachable state s of the collection, if $\langle s[M_1], t \rangle$ is an inner transition of M_1 , then $\langle s[M_2], t' \rangle$ is a legal transition of M_2 , where t' is the state in which the variables shared by M_1 and M_2 have the same values as in t and the rest of the variables of M_2 have the same values as in s . (We will refer to the latter condition by saying that M_2 *accepts all inner transitions of M_1*).

A collection of shared-state machines with security parameters is *secure* if and only if it is compatible, and whenever two machines share a variable, they assign the same equivalence relation to that variable.

The *composition* of a secure collection of shared-state machines with security parameters is the shared-state machine with security parameters defined as follows:

- The set of state variables is the union of the sets of state variables of machines in the collection.

- The set of labeled variables is the union of the sets of labeled variables of machines in the collection.
- The initial states of the composite are the common initial states of the collection.
- The state transitions of the composite are the common transitions of the collection.
- The inner transitions of the composite are all transitions made by taking an inner transition of a machine M in the collection and expanding it to a state transition of the composite by leaving all variables not of M fixed (such a state transition is legal for the composite by the definition of a compatible collection).
- The security parameters of the composite are the equivalence relations inherited from the machines in the collection (this is well-defined by the definition of a secure collection).

2.2.5 Composability

In this section we prove the following composition theorem for shared-state machines.

Theorem: The composite of a secure collection of shared-state restrictive machines is shared-state restrictive.

Proof: For each machine M in the collection, there exists an equivalence relation \simeq_M on the unlabeled states of M satisfying the conditions of restrictiveness. Let \simeq be the equivalence relation on the unlabeled states of the composite obtained by taking the conjunction of all of the \simeq_M 's. We will show that \simeq satisfies the conditions of restrictiveness for the composite.

Let \equiv be the equivalence relation on the states of the composite which is induced by \simeq . Let s_1 and s_2 be reachable states of the composite with $s_1 \equiv s_2$, and $\langle s_1, t_1 \rangle$ an inner transition of the composite. Obviously, s_1 and s_2 are common reachable states of the collection. By definition of the composite, there is some machine M in the collection such that $\langle s_1 \upharpoonright M, t_1 \upharpoonright M \rangle$ is an inner transition of M and all variables of the composite which are not

variables of M are the same in s_1 and t_1 . As remarked above, $s_1 \upharpoonright M$ and $s_2 \upharpoonright M$ are reachable states of M , and $s_1 \upharpoonright M \equiv_M s_2 \upharpoonright M$. By restrictiveness of M , $\langle s_1 \upharpoonright M, t_1 \upharpoonright M \rangle$ can be shifted to $s_2 \upharpoonright M$ with a shifting $\langle t_M^{(1)}, \dots, t_M^{(n)} \rangle$ consisting entirely of inner transitions of M . Let $t^{(i)}$ be the result of expanding $t_M^{(i)}$ to a common state of the collection with the assignments of s_2 . Thus, $t^{(1)} = s_2$. By a simple inductive argument, each $t^{(i)}$ is a common reachable state of the collection, and each transition in the sequence $\langle t^{(1)}, \dots, t^{(n)} \rangle$ is an inner transition of the composite. The last thing we must show is that $t^{(n)} \equiv t_1$. To do this, we must show two things:

1. **For each of the labeled variables v of the composite, $t^{(n)}(v)$ and $t_1(v)$ are equivalent by the security parameters of the composite.**

If v is not a variable of M , then $t^{(n)}(v) = t_1(v)$. If v is a variable of M , then $t^{(n)}(v) = t_M^{(n)}(v)$, which is equivalent (by the security parameters) to $t_1(v)$.

2. **For each machine M' in the collection, the unlabeled states of M' in $t^{(n)}$ and t_1 are equivalent by $\simeq_{M'}$.**

If $M' \neq M$, then the unlabeled states of M' in $t^{(n)}$ and t_1 are the same. If $M' = M$, then the unlabeled state of M' in $t^{(n)}$ is the unlabeled state of M' in $t_M^{(n)}$, which is $\simeq_{M'}$ to the unlabeled state of M' in t_1 .

■

2.2.6 Fundamental Security Theorem

The classical result to prove to show that a property is an information security property is to show that it implies some reasonable instantiation of deducibility security. For shared-state restrictiveness, it's not immediately obvious how to instantiate deducibility security because the equivalence relations define a notion of what the low view is, but don't define what the high information is that should not be deducible by low users. One possibility would be to adopt McCullough's preference functions [20] as the high information, and prove that instantiation of deducibility security. This would

probably be a useful and revealing exercise. We will prove a slightly different result than the classical one involving deducibility security.

Let M be a restrictive machine which is *closed*, that is, that has no outer state transitions. Let H be an equivalence relation on initial states of M such that for every pair of *initial* states s_0 and s_1 , there exists an *initial* state s_2 such that $H(s_0, s_2)$ and for every labeled variable V of M , $s_1(V) \simeq_V s_2(V)$, and for every unlabeled variable I of M , $s_1(I) = s_2(I)$ (where \simeq_V the equivalence relation on V). What this means is that the value⁶ of H is independent of both the unlabeled part of the state and the part of the state which is labeled “low”.

Let D be a function which takes a state of M and returns a set of H equivalence classes such that for any states s_0 and s_1 , if $s_0(V) \simeq_V s_1(V)$ for all labeled variables V , then $D(s_0) = D(s_1)$ (that is, D depends only on the part of the state which is labeled “low”).

Let \simeq be an equivalence relation on the unlabeled states of M satisfying restrictiveness and let \equiv be the induced equivalence relation on states of M .

We say that M is *accurate* if and only if for all initial states s_0 of M and all states s reachable from s_0 , $H(s_0) \in D(s)$ (that is, D can be interpreted as correct information about the initial value of H).

We say that M *says something about H* if and only if there exists a reachable state s of M such that $D(s)$ does not contain all the H -equivalence classes of initial states of M (that is, some possible initial value of H is ruled out by $D(s)$).

Theorem: If M is accurate, then M does not say anything about H .

Proof: Suppose M says something about H . Let s be a reachable state of M in which $D(s)$ does not contain all the H -equivalence classes of initial states of M . Let s_0 be an initial state such that $H(s_0) \notin D(s)$.

Let $\langle t_1, \dots, t_n \rangle$ be a sequence of states of M such that

- t_1 is an initial state of M .
- Each consecutive pair of states in the sequence form a legal (and there-

⁶We will abuse notation and also use H to represent the function which takes a state and returns the H -equivalence class of the state.

fore inner) transition of M

- $t_n = s$

From the fact that the initial value of H is independent of the low and unlabeled parts of the state, there exists an initial state t'_1 such that

- $H(t'_1) = H(s_0)$
- $t'_1(V) \simeq_V t_1(V)$ for all labeled variables V
- $t'_1(I) = t_1(I)$ for all unlabeled variables I

Thus, in particular, $t'_1 \equiv t_1$. By repeatedly shifting transitions in $\langle t_1, \dots, t_n \rangle$, we can construct a sequence $\langle t'_1, \dots, t'_{n'} \rangle$ such that each pair of consecutive states is a legal (and therefore inner) transition, and $t'_{n'} \equiv t_n = s$. Therefore, $t'_{n'}(V) \simeq_V s(V)$ for all labeled variables V , so $D(t'_{n'}) = D(s)$. Thus, $t'_{n'}$ is reachable from t'_1 , and $H(t'_1) = H(s_0) \notin D(s) = D(t'_{n'})$, so M is not accurate. ■

2.2.7 Afterword

2.2.7.1 Multiple Security Levels

For multiple security levels, we just need to have each of the labeled variables have one equivalence relation for each security level. The equivalence relation for a level l says when two values in the variable's type contain the same information of level $\leq l$. Restrictiveness for the machine for the entire level lattice is then just the conjunction of the definition of share state restrictiveness for the security parameters for each of the levels.

2.2.7.2 Interpreting the Fundamental Security Theorem

We think of H as describing some “secret” embedded in the initial state of a closed system. The value of this “secret” is assumed to be independent of

the unlabeled and “low” parts of the state, because otherwise there might be some part of the state labeled as “low” by the \equiv relation which can influence the value of H . This would allow the “low” entities within the system to conclude something about the value of H , but it would in effect be something which they had “written” into that value by constraining it, rather than something they had deduced. This is essentially the same reason for input-totality; if the thing which must be hidden is not free to vary through its set of possible values, then the classical meaning of “deduction” (being able to rule out some value) is not what we really want to mean by “information flow”. Shared-state restrictiveness allows inputs to be constrained, but the fundamental security theorem requires that whatever the “ultimate secret” that must be protected is, it must be unconstrained.

We think of D as a function which interprets some part of the labeled low part of the state of the closed system as a belief about the “secret”. For example, in an actual M , there might be some variable which ranges over some kind of expressions which stand for statements about the secret. D just takes the *entire* labeled low part of the state and extracts the beliefs about the secret that the low part of the closed system has collectively arrived at.

We are not dealing with probabilistic reasoning here. In probabilistic settings, it often makes sense to “guess” something about a secret which might be wrong, but which has a reasonable probability of being correct. In a nonprobabilistic setting like that of restrictiveness, however, there is no way of distinguishing between a “good” guess which might be wrong and a “bad” guess that might be wrong. There are only state transitions which may take you to a state in which you believe something false, and state transitions which do not. In the absence of such a distinction, we are essentially assuming that the low entities only want to make valid deductions, that is, they only want to conclude something about the “secret” if it is true. Thus, in this way of thinking, if M is not accurate, then we have successfully protected the secret. The fundamental security theorem says that if the low entities in the closed system limit themselves to valid deductions, they can never determine anything about the secret.

There is reason to believe that there is no real loss of generality in assuming that the secret is encoded in the *initial* state of M . There are two sources of information in a “run” of M which are not built into the design

(and therefore assumed known by the low entities). The first is the initial state, and the second is nondeterministic choices made during the run. The theorem above clearly covers the first source. However, it also covers the second, because, given any M , we could augment M by a variable which contains a sequence of McCullough's preference functions (even including one for the initial state), and augment the initial states and state transition relation so that all nondeterminism is resolved by the contents of the new variable. The new machine would have exactly the same possible behaviors of the original machine with respect to the original variables, but all sources of possible "secrets" would be packaged into the value in the new variable.

2.2.7.3 Why Only Inner Transitions?

A slightly suspicious feature of shared-state restrictiveness is that we only require *inner* transitions to be "shiftable". This reflects the assumption that outer transitions are caused by the environment, and that it is up to the environment to act securely. If the environment cannot be assumed to act securely, then the security of the system plus environment cannot be guaranteed. Also, in view of the composability theorem, if we keep hooking together restrictive machines until we have the machine and its environment included, we still have a restrictive machine, and that machine (if the entire environment has been included) is closed. In this case, restrictiveness will apply to all transitions, and the fundamental security theorem tells us that the result does not transmit information illegally.

2.3 Server/Client Restrictiveness

This section appeared as a paper in [52].

2.3.1 Introduction

Processes that make requests to shared resource handlers are often most naturally modeled with blocking reads. That is, such processes can be modeled as making a request and then waiting for the reply from the shared resource

handler. If we try to directly model such processes and then apply the restrictiveness security theory [25, 24], we may encounter a problem because the level of the reply is not directly connected to the level of the request. There is also a covert storage channel problem due to a high-level request potentially blocking a low-level activity. A number of methods have been developed to handle these problems, but those methods use models that are more complicated or less natural, or else use a weaker security property than restrictiveness.

For example, one way we can model shared resources is to use the method of [36, 49]. However, the expression of the models becomes complicated. This is because inputs are handled one at a time as separate transactions. For example, if one wants to read a file as part of some transaction one has to split the transaction into two pieces. The process does the first part of the transaction and terminates with a request for a file. The process then waits for the file to arrive as input and starts the second part of the transaction when that input is ready to be handled.

A method for handling states shared between machines has been developed by Ian Sutherland [58] and is described in section 2.2. That model, however, uses a different formulation of how processes communicate, and it is not as natural for describing input and output handling.

Another kind of composition of systems for shared resources is presented in [26]. In that paper, they weaken the restrictiveness security property in order to get the composition to work.

In this section, we present a natural method of decomposing a system into servers and clients and we identify security properties that they should satisfy. If the servers and clients satisfy these specified properties then the resulting combined system will be restrictive. (Unlike [26], the specified properties on the server and the client are not the same.) The method does not require splitting up a transaction as in [49]. It seems better suited for modeling communication with a shared file system than [58]. It also does not weaken the security property on the composite system as in [26].

The general security properties for the client and server can be complicated to work with. However, for a wide variety of models these properties can be significantly simplified. We will show how a variation of the methods of [49] can be used to make this simplification.

Similar results could be obtained for other security properties similar to restrictiveness. However, this work uses non-deterministic models as a step in the proof of the composition theorem and so it will not be directly applicable to non-interference [14].

The remainder of this section is divided into two parts. In section 2.3.2 we describe restrictiveness, the decomposition, and the properties that the parts should satisfy to make the composite system restrictive. We then prove the composition theorem. In section 2.3.3 we describe how, for many models, these properties can be simplified and how the analysis can be performed, using methods similar to [49].

2.3.2 Theory

2.3.2.1 Restrictive state machines

The definition of restrictiveness presented in this section is taken from [51], but it is essentially the same as [25].⁷ This section provides only the basic notation and definition of restrictiveness. For an introduction to restrictiveness, see [25].

A *state machine* is a set of states, an initial state (or set of states), a set of events, and a transition relation. The events may be inputs, outputs, or internal transitions. The transition relation may be non-deterministic. A typical transition will be represented as $s_1 \xrightarrow{\alpha} s_2$, that is, state s_1 goes to state s_2 by the event sequence α . For a *rated state machine* we also have a level function, which assigns a security level to each event (the security levels are a set with a partial order). We will use the word *lev* to refer to the level function. In this discussion all of the state machines will be rated.

For each security level ℓ , we define an equivalence relation on the states, usually denoted as \approx_ℓ . Intuitively, this relation defines when two states appear the same to someone who has clearance only up to level ℓ . An event is said to be *low* with respect to some ℓ if the level of the event is less than or equal to ℓ . An event is said to be *high* with respect to some ℓ if it is not low.

If v is a subset of events and γ is a sequence of events, then we will denote

⁷Also given in section 2.1.2.1.

that subsequence of γ restricted to events from v by $\gamma \upharpoonright v$. We will use the notation $\gamma \upharpoonright \ell$ to mean the subsequence of γ restricted to the events that are low with respect to ℓ .

The empty sequence will be written as $\langle \rangle$. The concatenation of two sequences is represented by \wedge . The concatenation of a singleton event sequence $\langle e \rangle$ and another sequence l will be represented as $e \wedge l$.

For a state machine with security relation \approx_ℓ as above, we say that it is *restrictive* if

1. for every level ℓ , for all states s_1, s'_1, s_2 such that $s_1 \approx_\ell s_2$, and for all input sequences β_1 and β_2 such that $s_1 \xrightarrow{\beta_1} s'_1$ and $\beta_1 \upharpoonright \ell = \beta_2 \upharpoonright \ell$, there exists a state s'_2 such that

- (a) $s_2 \xrightarrow{\beta_2} s'_2$
- (b) $s'_2 \approx_\ell s'_1$.

A state machine that satisfies this condition is input-total except for possibly the lowest level events. That is, the machine can always accept a sequence of non-lowest level inputs. (Just choose $\beta_1 = \langle \rangle$ and $s'_1 = s_1 = s_2$.)

It also follows that high-level inputs leave the low-level view of the state (i.e., the equivalence class under \approx_ℓ) unchanged, as can be seen by letting β_1 be a sequence of high level inputs and letting β_2 be $\langle \rangle$.

2. for every level ℓ , for all states s_1, s'_1, s_2 such that $s_1 \approx_\ell s_2$, and for all noninput events e such that $s_1 \xrightarrow{e} s'_1$, there exist a state s'_2 and a sequence γ of noninputs such that

- (a) $s_2 \xrightarrow{\gamma} s'_2$
- (b) $s'_2 \approx_\ell s'_1$
- (c) $\gamma \upharpoonright \ell = e \upharpoonright \ell$.

2.3.2.2 Shared resource decomposition

In general, we are interested in how to securely compose a system consisting of processes that make blocking requests and processes that handle those

requests. For simplicity we will initially assume that the decomposition involves only two process. We will call the process that handles the requests (e.g., manages some resource) the server and the property that it should satisfy *server restrictiveness*. We will call the process that may make blocking requests on the server the client process and the property that it should satisfy *client restrictiveness*.

Suppose C is a client process and S is a server process. We will partition the events of these processes as follows: E_{CS} are those events from C causing C to block on a reply from S , E_{SC} are those events from S to C in response to a blocking request, E_S are events of S that are not in $E_{SC} \cup E_{CS}$, and E_C are events of C not in $E_{CS} \cup E_{SC}$. (Remark: We do not exclude the possibility that C and S may communicate in nonblocking ways. However, in a composition of C and S they must both use the same definitions of E_{CS} and E_{SC} .)

We will prove that a proper composition of a server restrictive process and a client restrictive process is restrictive. In section 2.3.2.6 we will describe how this can be generalized to multiple clients and servers.

Client restrictiveness Client restrictiveness is much like restrictiveness, but it differs in how unblocking responses from the server are handled. Instead of directly analyzing a client we will transform it into a similar process, where the replies from a server are replaced with nondeterministic transitions. Replies that are clearly not secure will not be allowed in the transformed machine. We will then have to strengthen the restrictiveness property on the transformed machine, because not all of its transitions will really be possible in a composite system.

First we describe the construction of the new machine. For each occurrence of a request followed by inputs that are not unblocking and then followed by receiving an unblocking reply from the server, we will replace the unblocking replies by a collection of “allowable” nondeterministic transitions based on the level of that request. (The reason for including intervening inputs that are not unblocking is that we want the blocked process to continue to buffer inputs.)

A replaceable occurrence in the *Client* is any state c_1 where o is a blocking request, β is a sequence of inputs that are not unblocking, and

c_1 has the collection of transitions $c_1 \xrightarrow{o \wedge \beta} c_2 \xrightarrow{i} c_3(i)$ for each possible unblocking reply i from the server. We also require that the only transitions to c_2 end with a blocking request at level o followed by some sequence of inputs that are not unblocking, and that the only events to $c_3(i)$ are unblocking replies. (Expressed in a process algebra syntax the transitions are $send(o); receive_{lev(o)}(fromserver, \lambda i. c_3(i))$ in parallel with some input buffering mechanism.)

This occurrence will be replaced by a collection of transitions $c_1 \xrightarrow{o \wedge \beta} c_2 \xrightarrow{\tau} c_3(i)$ for each i where $lev(i) \geq lev(o)$ and where τ is a silent transition (i.e., τ is a delay or internal event). In other words, the potential unblocking input transitions are replaced by non-deterministic transitions to a state that could be arrived at by some input from an arbitrary server behaving securely. (Or equivalently, $send(o); Select(J, \lambda j. c_3(j))$ in parallel with some input buffering mechanism, where J is the set of events to the *Client* whose level is greater than or equal to the level of o .)

Call this transformation function, which replaces the input from the server with nondeterminism, *Change*.

Client restrictiveness is a strengthening of restrictiveness on $Change(Client)$. Let us call the transformed machine $Change(Client)$, $Client'$. Notice that the states of $Client'$ are the same as those of the *Client*, only the transitions have changed.

$Client'$ will have more potential behaviors than the *Client* in a composite system, because not all of the non-deterministic responses of $Client'$ will really be possible responses of the *Server* of the composite system. For this reason we will have to impose extra conditions on $Client'$ besides restrictiveness to make sure the composite system will be restrictive.

For client restrictiveness, we require the following:

1. The *Client* process should be of the right form. Blocking requests to the server are immediately followed by a sequence of inputs that are not unblocking, and then an unblocking reply from the server. Unblocking replies from the server are immediately preceded by a blocking request followed by sequences of inputs that are not unblocking.

2. The state machine *Client'* satisfies restrictiveness.
3. We will make a slightly stronger constraint on low blocking outputs to the server than is provided by restrictiveness, because the extra non-determinism of *Client'* will not be available in the combined machine.

For every level ℓ , for all client states c_1, c'_1, c_2 such that $c_1 \approx_\ell c_2$, and for all low blocking outputs e to the server such that $c_1 \xrightarrow{e} c'_1$, there exist a state c'_2 and a sequence γ' of noninputs such that

- (a) $c_2 \xrightarrow{\gamma' \wedge e} c'_2$
- (b) $c'_2 \approx_\ell c'_1$
- (c) $\gamma' \uparrow \ell = \langle \rangle$.

The reason why this is stronger than restrictiveness is that the output property of restrictiveness permits choosing a sequence with high-level outputs after e .

4. For each transformed input from the server we will need to assure that high-level requests will cause only an effect to the high-level state.

If o is a high-level request to the server with $c_1 \xrightarrow{o \wedge \beta} c_2 \xrightarrow{\tau} c_3(i)$ a transition sequence of *Client'*, we require that $c_3(i') \approx_\ell c_2$.

5. We need a condition similar to the restrictiveness input condition for states making nondeterministic transitions in place of server responses.

If $c_1 \approx_\ell c_2$ and if i and j are nondeterministic events introduced by *Change* such that $c_1 \xrightarrow{i} c_1(i)$, $c_2 \xrightarrow{j} c_2(j)$, and $\langle i \rangle \uparrow \ell = \langle j \rangle \uparrow \ell$, then $c_1(i) \approx_\ell c_2(j)$.

6. We will not allow equivalence relations where a state that is blocked after sending a low-level request is equivalent to a nonblocked state.

If o is a low-level request to the server, β_1 is a sequence of inputs, $c_1 \xrightarrow{o \wedge \beta_1} c_2 \xrightarrow{\tau} c_3(i)$, and $t_2 \approx_\ell c_2$, then the transitions from t_2 are the nondeterministic transitions introduced by *Change* and inputs.

7. Restrictiveness involves a liveness constraint, which requires some added constraint on *Client'*. We require that for every state of *Client'* there is a bound on how many times the server can be called from that state, without an intervening input. More formally, we require that there be a function from states into ordinals, which we call *rank*, such that

- (a) for all noninput events e , if $c_1 \xrightarrow{e} c_2$ then $\text{rank}(c_1) \geq \text{rank}(c_2)$, and
- (b) for every request o to the server, if $c_1 \xrightarrow{o} c_2 \xrightarrow{\tau} c_3(i)$ then $\text{rank}(c_2) > \text{rank}(c_3)$.

Note: We can typically build a rank function that assigns natural numbers to states and decreases by one on each return from the server.

We will call these properties (CR1-CR7).

Server restrictiveness A server process needs to satisfy a property stronger than restrictiveness because it must properly respond to blocking requests. (Not all inputs need be blocking requests from the client.)

For server restrictiveness, we require the following:

1. The Server process should be of the right form.

- (a) Every blocking request can get a response.

For each pair of states s_1 and s_2 , blocking request from the client e , and γ_1 containing events which are not unblocking, such that

$s_1 \xrightarrow{e \wedge \gamma_1} s_2$, there is a state s_3 , a noninput sequences of events not to the client γ_2 , and an unblocking event e' to the client such that

$$s_2 \xrightarrow{\gamma_2 \wedge e'} s_3.$$

- (b) An unblocking response to the client can be made only after a blocking request.

If $s_1 \xrightarrow{\gamma \wedge e} s_2$, where e is an unblocking output to the client and γ contains no blocking requests from the client, then there is a blocking request from the client e' , noninput sequence γ' containing no

other replies to the client, and a “ready” state s_0 (i.e., a state such that there does not exist a noninput sequence γ'' containing an unblocking output with $s_0 \xrightarrow{\gamma''} s_3$), so that if

$s_0 \xrightarrow{e' \wedge \gamma'} s_1 \xrightarrow{\gamma \wedge e} s_2$, then the number of blocking requests of $e' \wedge \gamma'$ is greater than the number of unblocking outputs of γ' . (In fact, in the parallel composition, the client will not make multiple requests without an intervening reply, so this condition will say that there are no requests and no replies in γ' .)

2. The Server must be restrictive.
3. For states s_1 and s_2 , high blocking request e from the client, and γ_1 containing no unblocking replies where $s_1 \xrightarrow{e \wedge \gamma_1} s_2$, there is a state s_3 and high noninput sequences of events γ_2, γ_3 , and a high unblocking event to the client e' such that $s_2 \xrightarrow{\gamma_2 \wedge e' \wedge \gamma_3} s_3$ with $s_2 \approx_\ell s_3$.

This condition requires that on high-level blocking requests from the client, the server could complete that request invisibly. The technical reasons for this requirement and the implications are discussed in section 2.3.2.7.

4. On low-level requests the equivalence relation should make a distinction between whether a high response to that request has been made, since responses will unblock the client.

If e is a low-level blocking request from the client, and if $s_1 \xrightarrow{e \wedge \gamma_1} s'_1 \xrightarrow{\gamma'_1} s''_1$ where γ_1 does not contain an unblocking reply to the server and γ'_1 contains a high-level unblocking reply with no blocking requests, and if $s'_2 \xrightarrow{\gamma'_2} s''_2$ where $s'_1 \approx_\ell s'_2$ and $s''_1 \approx_\ell s''_2$, then γ'_2 contains an unblocking reply.

We will call these properties (SR1-SR4).

2.3.2.3 Composition theorem

In this section we will show that the composition of a client restrictive process and a server restrictive process is a restrictive process.

The parallel composition of these machines $R=C||S$ is defined as follows. We define the set of potential states of R as ordered pairs from C and S . We will use the notation $c||s$ to refer to a potential state. Some of these pairs do not make sense. In particular, if c is a state of C that is blocked waiting for a return value from S after a request e , then for $c||s$ to make sense s must be a state that can be arrived at from some s' by some transition $e^\wedge\gamma$ where γ has no other blocking requests or unblocking replies. Also, if c is a state of C that is not blocked for a read, then for $c||s$ to make sense s cannot be a state that could eventually send an unblocking output to the client (without an intervening request). Pairs that satisfy these conditions will be called good potential states and pairs that don't will be called bad potential states. If S is server restrictive and C is client restrictive, these bad potential states are not reachable from good potential states. The states of R consist of just the good potential states.

The input and output events are the input and output events from $E_S \cup E_C$ (defined in the beginning of section 2.3.2.2), which are not events from S to C or from C to S . (We have not excluded the possibility that the server and client may communicate through nonblocking requests.) For any e in E_S that is not to or from C , c_1 a state of C and transition of S , $s_1 \xrightarrow{e} s_2$, we have $c_1||s_1 \xrightarrow{e} c_1||s_2$. For any e in E_C that is not to or from S , s_1 a state of S , and transition of C , $c_1 \xrightarrow{e} c_2$, we have $c_1||s_1 \xrightarrow{e} c_2||s_1$.

If e is an event from S to C or from C to S (including events from E_{CS} or E_{SC}) and we have $c_1 \xrightarrow{e} c_2$ and $s_1 \xrightarrow{e} s_2$, then $c_1||s_1 \xrightarrow{\tau} c_2||s_2$. (Note that τ , an internal transition, is used since the effect is not to receive inputs or produce outputs to the external environment of R .)

Before we begin the composition theorem we show three lemmas that we will use to convert sequences of events in $Client'$ into sequences of events for R . The reason this result is not trivial is that not all of the nondeterministic responses of $Client'$ may be possible in the combined machine.

Lemma 1: Suppose R is a parallel composition of a client restrictive

and server restrictive process as described above. For any $c||s$ that is a state of R such that $c \xrightarrow{\gamma} c'$ where γ is a sequence of noninput events of $Client'$ with no low-level events to the server and not starting with an unblocking transition based on a low-level request, there is a γ' of R such that

1. $c||s \xrightarrow{\gamma'} c''||s'$
2. $c'' \approx_\ell c'$, $s \approx_\ell s'$, and
3. $\gamma' \upharpoonright \ell = \gamma \upharpoonright \ell$ (when events between the server and client are replaced by τ).

Proof: We prove the result by induction on the rank of c . Pick any c, s, c', γ as in the hypothesis of the lemma.

1. Suppose γ does not contain any unblocking events. Since it is a noninput sequence of $Client'$, the only events of γ that pertain to the server are high level outputs to the server. Let δ be the restriction of γ to these events. By restrictiveness of the server (SR2) we can find a s' such that $s \xrightarrow{\delta} s'$ with $s \approx_\ell s'$. Then just let $c'' = c'$ and we have $c||s \xrightarrow{\gamma'} c''||s'$ with the desired properties.
2. Otherwise γ contains an unblocking reply from the server.

We can express $c \xrightarrow{\gamma} c'$ as $d_1 \xrightarrow{\gamma_1} d_2 \xrightarrow{e_1} d_3 \xrightarrow{\tau} d_4 \xrightarrow{\gamma_2} c'$, where c is one of d_1, d_2 , or d_3 , e_1 is an event to the server, $d_3 \xrightarrow{\tau} d_4$ is one of the non-deterministic transitions introduced by *Change*, and γ_1 contains no blocking or unblocking events. Note that γ_1 and γ_2 may be empty.

If c is d_1 or d_2 as in the case above, we have $d_1||s \xrightarrow{\gamma_1} d_2||s \xrightarrow{e_1} d_3||s'''$ for some $s''' \approx_\ell s$. If c is d_3 , then just let $s''' = s$. In either case we have $d_3||s'''$ with $s''' \approx_\ell s$. We still have to adjust the rest of the transition.

In the case that c is d_3 , by assumption the internal transition is made based on a high-level request. Otherwise, e will be high since γ contains no low-level events to the server. Hence, in any case we have that d_3 is a state blocked because of a high request.

By server restrictiveness (SR3), there is a transition, $\delta_1 \wedge e_2 \wedge \delta_2$, where δ_1 and δ_2 are noninput sequences of the server with events from E_S and e_2 is in E_{SC} , such that $s''' \xrightarrow{\delta_1 \wedge e_2 \wedge \delta_2} s''$ and such that all of the events of this transition are high and such that $s''' \approx_\ell s''$. Some of the events of δ_1 and δ_2 may be inputs (which are not unblocking) to the client. Let δ'_1 and δ'_2 be the restrictions of these sequences to events to the *Client*. As they are high, by (CR2) we can find a d'_3 such that $d_3 \xrightarrow{\delta'_1} d'_3$ with d'_3 equivalent to d_3 . Since the request e_1 was high by (CR4) we have $d'_3 \xrightarrow{\tau} d''_3$ (where τ is the internal transition of *Client'* used for e_2) and $d''_3 \approx_\ell d'_3$. And then we can find a d'''_3 with $d''_3 \xrightarrow{\delta'_2} d'''_3$ with d'''_3 equivalent to d''_3 . This adjustment does not change the low-level view of γ in the composite machine. We also have $d'''_3 \approx_\ell d_3$. Again, by (CR4) $d_3 \approx_\ell d_4$, so we have $d'''_3 \approx_\ell d_4$. Let $d'_4 = d'''_3$. We have $d_3 || s''' \xrightarrow{\delta_1 \wedge e_2 \wedge \delta_2} d'_4 || s''$ with $d_4 \approx_\ell d'_4$ and $s'' \approx s'''$. All of the added events of this sequence are high (and the τ of γ corresponds to the internal transition of R , e_2 .) We still need to adjust the last part of γ .

By restrictiveness of *Client'* (CR2), there is a γ'_2 with $d'_4 \xrightarrow{\gamma'_2} c''$ with $c'' \approx_\ell c'$ and $\gamma'_2 \uparrow \ell = \gamma_2 \uparrow \ell$. Note that this means γ'_2 also contains no low-level requests to the server. By (CR1) and (SR1) γ'_2 cannot start with an unblocking transition.

Now d'_4 has a lower rank than c_2 (since c_2 is one of d_1, d_2, d_3), so by the induction hypothesis we can find a γ''_2 that is high such that $d'_4 || s'' \xrightarrow{\gamma''_2} c''' || s'$ with $c''' \approx_\ell c''$, $s' \approx_\ell s''$, and $\gamma''_2 \uparrow \ell = \gamma'_2 \uparrow \ell$. So we have $c''' \approx_\ell c'$, and hence $s \approx_\ell s'$ and $\gamma''_2 \uparrow \ell = \gamma_2 \uparrow \ell$.

Hence we have $d_1 || s \xrightarrow{\gamma_1} d_2 || s \xrightarrow{e_1} d_3 || s''' \xrightarrow{\delta_1 \wedge e_2 \wedge \delta_2} d'_4 || s'' \xrightarrow{\gamma''_2} c''' || s'$, with $c''' \approx_\ell c'$ and $s' \approx_\ell s$, and the entire sequence of events restricted to l is equal to $\gamma \uparrow \ell$. (Or, in case the sequence starts at d_3 , $d_3 || s \xrightarrow{\delta_1 \wedge e_2 \wedge \delta_2} d'_4 || s'' \xrightarrow{\gamma''_2} c''' || s'$.)

□

Lemma 2: Suppose R is a parallel composition of a client restrictive and server restrictive process as described above. For any $c||s$ that is a state of R such that $c \xrightarrow{\gamma} d \xrightarrow{e} c'$, where γ is a sequence of noninput events of $Client'$ with no low-level events to the server and not starting with an unblocking transition based on a low-level request, and e is a low blocking event to the server, there is a γ' such that

1. $c||s \xrightarrow{\gamma'} d||s'' \xrightarrow{e} c'''||s'$
2. $c''' \approx_\ell c'$
3. $s \approx_\ell s'''$
4. $\gamma' \upharpoonright \ell = \gamma \upharpoonright \ell$ (when events between the server and client are replaced by τ).

Proof: We prove this result by induction on the rank of c . Pick c, c', s, d, γ, e to meet the hypotheses so that the lemma is true for all c 's of lower rank.

If there are no unblocking events in γ , then as in the proof of Lemma 1, we have that $c||s \xrightarrow{\gamma} d||s'''$ for some $s''' \approx_\ell s$. So, since the next transition of d is by e and $d||s'''$ is a good state, there is an s' such that $d||s''' \xrightarrow{e} c'||s'$.

Otherwise we can use Lemma 1 for the γ part to obtain a d' and a sequence γ' such that $c||s \xrightarrow{\gamma'} d'||s''$ in R where γ' corresponds to γ , $d \approx_\ell d'$, and $s'' \approx_\ell s$. Since γ contains an unblocking event, by the construction of Lemma 1, so will γ' and hence, d' will have a rank less than c .

Since $d' \approx_\ell d$ and $d \xrightarrow{e} c'$ by (CR3), we have a c'' with $d' \xrightarrow{\gamma'' \wedge e} c''$, $c'' \approx c'$, and $\gamma'' \upharpoonright \ell = \langle \rangle$. As d' is of lower rank, we can apply the inductive hypothesis to obtain $d'||s'' \xrightarrow{\gamma'''} d||s''' \xrightarrow{e} c'''||s'$ in R where $c''' \approx_\ell c''$, $s''' \approx_\ell s''$, and $\gamma''' \upharpoonright \ell = \langle \rangle$. Hence $c''' \approx_\ell c'$ and $s''' \approx_\ell s$.

□

Lemma 3: Let R be a parallel composition of a client restrictive and server restrictive process as described above. Suppose $c_1||s_1$ is a state of R

such that $c_1 \xrightarrow{\gamma_1} d_1 \xrightarrow{e} d_2 \xrightarrow{\gamma_2} c_2$, where γ_1 and γ_2 are sequences of noninput events of *Client'* with no low-level events to the server and such that γ_1 does not start with an unblocking transition based on a low-level request, and e is a low event to the server which is not blocking. Further suppose that $s_1 \xrightarrow{e} s_2$. Then there is a γ'_1 , a γ'_2 , and a s'_2 such that

1. $c_1 || s_1 \xrightarrow{\gamma'_1} d'_1 || s'_1 \xrightarrow{e} d_2 || s''_2 \xrightarrow{\gamma'_2} c'_2 || s'_2$
2. $c'_2 \approx_\ell c_2$
3. $s'_2 \approx_\ell s_2$
4. $\gamma'_1 \upharpoonright \ell = \gamma_1 \upharpoonright \ell$ and $\gamma'_2 \upharpoonright \ell = \gamma_2 \upharpoonright \ell$ (when events between the server and client are replaced by τ).

Proof: We prove this result by induction on the rank of c . Pick $c_1, d_1, d_2, c_2, s_1, s_2, \gamma_1, \gamma_2, e$ to meet the hypotheses so that the lemma is true for all c_1 's of lower rank.

If there are no unblocking events in γ_1 , then as in the proof of Lemma 1, we have that $c_1 || s_1 \xrightarrow{\gamma_1} d_1 || s'_1$ for some $s'_1 \approx_\ell s_1$. By (SR2) we can find a state s''_2 such that $d_1 || s'_1 \xrightarrow{e} d_2 || s''_2$ and $s''_2 \approx_\ell s_2$. Now γ_2 does not start with an unblocking response, since e is not a blocking request. Hence we can apply Lemma 1 to γ_2 and find c'_2, s'_2 and γ'_2 so that $d_2 || s''_2 \xrightarrow{\gamma'_2} c'_2 || s'_2$, with $c'_2 \approx_\ell c_2$ and $s'_2 \approx_\ell s_2$.

Otherwise we can use Lemma 1 for the γ_1 part to obtain a d'_1 and a sequence γ'_1 such that $c_1 || s_1 \xrightarrow{\gamma'_1} d'_1 || s'_1$ in R , where γ'_1 corresponds to γ_1 , $d_1 \approx_\ell d'_1$, and $s'_1 \approx_\ell s_1$. Since γ_1 contains an unblocking event, by the construction of Lemma 1 so will γ'_1 and hence, d'_1 will have a rank less than c_1 .

Since $d'_1 \approx_\ell d_1$ and $d_1 \xrightarrow{e} d_2$, by (CR2) we have a d'_2 with $d'_1 \xrightarrow{\gamma_3 \wedge e \wedge \gamma_4} d'_2$, $d'_2 \approx_\ell d_2$, $\gamma_3 \upharpoonright \ell = \langle \rangle$, and $\gamma_4 \upharpoonright \ell = \langle \rangle$. Note that from d'_1 , γ_3 cannot start with an unblocking event based on a low level request, by the construction of γ'_1 . Again, by (CR2) we can find a γ'_2 such that $d'_2 \xrightarrow{\gamma'_2} c'_2$, with $c'_2 \approx_\ell c_2$ and $\gamma'_2 \upharpoonright \ell = \gamma_2 \upharpoonright \ell$. So we have $d'_1 \xrightarrow{\gamma_3 \wedge e \wedge \gamma_4 \wedge \gamma'_2} c'_2$.

Since $s'_1 \approx_\ell s_1$, we can find an s'_2 with $s'_1 \xrightarrow{e} s'_2$ and $s'_2 \approx s_2$. As d'_1 is of lower rank, we can apply the inductive hypothesis with $d'_1 || s'_1$ and obtain the desired result.

□

We are now ready to prove the main result.

Theorem: If C is client restrictive, S is server restrictive, and R is the parallel composition of C and S as defined above, then R is restrictive.

Proof: For the machine R we will use an equivalence relation on states used in the restrictiveness proofs of the parts. Let $Client'$ be the machine $Change(C)$. We will also use the abbreviation D for $Client'$. For D , we can use the same state names as for C but some of the transitions may be different (see definition of $Change$ above). Let $\approx_{\ell,D}$ and $\approx_{\ell,S}$ be the equivalence relations used for showing restrictiveness of D and S . Let \approx_ℓ on R be defined as

$$c_1 || s_1 \approx_\ell c_2 || s_2 \text{ if and only if } c_1 \approx_{\ell,D} c_2 \text{ and } s_1 \approx_{\ell,S} s_2.$$

When using the equivalence relations we will drop the D and S from the notation for \approx_ℓ , since they can be inferred from the context.

The proof of restrictiveness of R naturally falls into two obligations — showing the input and the output properties.

2.3.2.4 Input property

The proof of the input property is straightforward.

Pick any level ℓ and any states r_1, r'_1, r_2 of R , such that $r_1 \approx_\ell r_2$, and any input sequences β_1 and β_2 such that $r_1 \xrightarrow{\beta_1} r'_1$ and $\beta_1 \upharpoonright \ell = \beta_2 \upharpoonright \ell$.

We need to show that there exists a state r'_2 of R such that

1. $r_2 \xrightarrow{\beta_2} r'_2$
2. $r'_2 \approx_\ell r'_1$.

Since $R = S || C$, there are states $s_1, s'_1, s_2, c_1, c'_1, c_2$ such that $r_1 = s_1 || c_1$, $r'_1 = s'_1 || c'_1$, and $r_2 = s_2 || c'_2$.

By restrictiveness of S there exists an s'_2 meeting:

1. $s_2 \xrightarrow{\beta_2 \upharpoonright (E_S \cup E_{CS})} s'_2$ (Recall that \upharpoonright is just the restriction of the event sequence to some subset.)
2. $s'_2 \approx_\ell s'_1$.

Since β_2 is just a sequence of inputs for R , it contains no events from E_{CS} , and so the first condition is just $s_2 \xrightarrow{\beta_2 \upharpoonright E_S} s'_2$.

For the client process C , we have assumed it is client restrictive so that D is restrictive.

Notice that input transitions from E_C are the same for both D and C (only the transitions from E_{SC} are changed). Further, the input sequences of R do not include events from E_{SC} . Hence we still have in D that $c_1 \xrightarrow{\beta_1 \upharpoonright E_C} c'_1$.

By restrictiveness of D there exists a c'_2 meeting:

1. $c_2 \xrightarrow{\beta_2 \upharpoonright E_C} c'_2$
2. $c'_2 \approx_\ell c'_1$.

So in R we can let $r'_2 = c'_2 || s'_2$ where c'_2 and s'_2 are chosen as above. Then

1. $r_2 \xrightarrow{\beta_2} r'_2$ since β_2 has input events from the disjoint sets E_C and E_S , and
2. $r'_2 \approx_\ell r'_1$ by the definition of \approx_ℓ .

2.3.2.5 Output property

Pick any level ℓ and any states r_1, r'_1, r_2 of R such that $r_1 \approx_\ell r_2$, and pick any noninput event e such that $r_1 \xrightarrow{e} r'_1$.

We must show there exists a state r'_2 and a sequence γ of noninputs such that

1. $r_2 \xrightarrow{\gamma} r'_2$
2. $r'_2 \approx_\ell r'_1$
3. $\gamma \uparrow \ell = e \uparrow \ell$.

Since $R = S||C$, we have $s_1, s'_1, s_2, c_1, c'_1, c_2$ such that $r_1 = c_1||s_1$, $r'_1 = c'_1||s'_1$, and $r_2 = c_2||s_2$.

We prove the result by cases, depending on whether e is from the client or server.

1. Case: e is in E_C or E_{CS} .

We divide this case into subcases:

- (a) Case: e is not a low-level event to the server.

By the definition of \approx_ℓ we have $c_1 \approx_\ell c_2$. Note that for e in E_C or E_{CS} , $c_1 \xrightarrow{e} c'_1$ is a transition of D .

Choose γ and c'_2 to satisfy

- i. $c_2 \xrightarrow{\gamma} c'_2$
- ii. $c'_2 \approx_\ell c'_1$
- iii. $\gamma \uparrow \ell = e \uparrow \ell$.

Notice that γ contains no low-level events to the server. It cannot start with an unblocking transition based on a low-level request, because otherwise by (CR6) c_1 must also be a blocked state. But that is not possible by the case assumption that e is in E_C or E_{CS} . So by Lemma 1 we can extend it to a transition in R . That is, there exist c''_2, γ' such that

- i. $c_2||s_2 \xrightarrow{\gamma'} c''_2||s'_2$
- ii. $c''_2 \approx_\ell c'_2$
- iii. $s'_2 \approx_\ell s_2$
- iv. $\gamma' \uparrow \ell = \gamma \uparrow \ell$ with the appropriate hiding of internal events (and hence $\gamma' \uparrow \ell = e \uparrow \ell$).

If e is not to the server then we have $s'_1 = s_1$. Else by case hypothesis it is a high input to the server. Then by restrictiveness of the server (SR2), we must have $s'_1 \approx_\ell s_1$ (it must be equivalent to a state where no inputs have arrived). So in either case we have $s'_1 \approx_\ell s_1$.

We have $s_2 \approx_\ell s_1$ by hypothesis and we have just found an s'_2 with $s'_2 \approx_\ell s_2$. So $s'_2 \approx_\ell s'_1$. Then we have $c'_2 || s'_2 \approx_\ell c'_1 || s'_1$ as desired.

(b) Case: e is a low-level blocking event to the server.

Since e is in E_{CS} , by (CR3), we can choose γ and c'_2 to satisfy

- i. $c_2 \xrightarrow{\gamma \wedge e} c'_2$
- ii. $c'_2 \approx_\ell c'_1$
- iii. $\gamma \upharpoonright \ell = \langle \rangle$.

γ does not contain any low requests to the server, and it cannot start with an unblocking transition based on a low-level input by (CR6). By Lemma 2, we have $c_2 || s_2 \xrightarrow{\gamma'} d || s''_2 \xrightarrow{e} c''_2 || s'''_2$ with $c''_2 \approx_\ell c'_2$, $s''_2 \approx_\ell s_2$, and $\gamma' \upharpoonright \ell = \gamma \upharpoonright \ell = \langle \rangle$. Since $s_1 \xrightarrow{e} s'_1$, by restrictiveness of the server (SR2) we can find an s'_2 such that $s_2 \xrightarrow{e} s'_2$ and $s'_2 \approx_\ell s'_1$. Hence we also have $c_2 || s_2 \xrightarrow{\gamma'} d || s''_2 \xrightarrow{e} c''_2 || s'_2$, which is the desired result.

(c) Case: e is a low-level event to the server that is not blocking.

By (CR2), we can choose γ and c'_2 to satisfy:

- i. $c_2 \xrightarrow{\gamma_1 \wedge e \wedge \gamma_2} c'_2$
- ii. $c'_2 \approx_\ell c'_1$
- iii. $\gamma_1 \upharpoonright \ell = \langle \rangle$ and $\gamma_2 \upharpoonright \ell = \langle \rangle$.

Say, $c_2 \xrightarrow{\gamma_1 \wedge e} d \xrightarrow{\gamma_2} c'_2$.

γ_1 and γ_2 do not contain any low requests to the server, and γ_1 cannot start with an unblocking transition based on a low-level input by (CR6). Hence, we can apply Lemma 3 and we get the desired result.

2. Case: e is an event in E_S or E_{SC} .

(a) Case: e is not a low-level unblocking output to the *Client*.

We have $c_1 || s_1 \xrightarrow{e} c'_1 || s'_1$.

By restrictiveness of the server (SR2) we can find $s_2 \xrightarrow{\gamma} s'_2$ with $\gamma \uparrow \ell = \langle e \rangle \uparrow \ell$ and $s'_2 \approx_\ell s'_1$.

(i) Suppose γ contains no unblocking events to the client. Then, by (SR4), e cannot be a reply made on the basis of a low-level request from the server. If e is not an output to the *Client* then we have $c_1' = c_1$ and hence $c'_1 \approx_\ell c_1$. If e is an unblocking reply then since it is due to a high request, by (CR4) we have $c'_1 \approx_\ell c_1$. So in these two cases we can use (CR2) on the events of γ and the fact that γ does not contain an unblocking reply to choose $c'_2 \approx_\ell c'_1$ so that $c_2 || s_2 \xrightarrow{\gamma} c'_2 || s'_2$, and we get the desired transition. Otherwise, e is an output to the *Client* which is not unblocking. Then by applying restrictiveness to the events of γ that are inputs to the *Client* (there are no unblocking replies), we can just choose $c'_2 \approx_\ell c_1$ so that, $c_2 || s_2 \xrightarrow{\gamma} c'_2 || s'_2$, and we get the desired transition.

(ii) Suppose γ does contain an unblocking reply. We then know that γ is of the form $\gamma_1 \wedge e' \wedge \gamma_2$ where e' is an unblocking reply to the *Client* and γ_1, γ_2 contain no other unblocking replies by (SR1), (CR1). e' must be high, since e is not a low unblocking event to the server and $\gamma \uparrow \ell = e \uparrow \ell$.

c_2 must be ready to receive a reply since $c_2 || s_2$ is a state of R .

If e is not an unblocking reply to the *Client*, then by (SR4) e' cannot be made on the basis of a low-level request. Let γ'_1 and γ'_2 be the inputs to the *Client* from γ_1 and γ_2 . By (CR2) choose c''_2

so that $c_2 \xrightarrow{\gamma'_1} c''_2$ with $c''_2 \approx_\ell c_2$ or $c''_2 \approx_\ell c'_1$ in the case that e is low and in γ'_1 . By (CR4), we have $c''_2 \xrightarrow{\tau} c'''_2$ (where τ is the internal transition corresponding to e') with $c''_2 \approx_\ell c'''_2$. Again by (CR2)

choose c'_2 so that $c'''_2 \xrightarrow{\gamma'_2} c'_2$ with $c'_2 \approx_\ell c'''_2$ or $c'_2 \approx_\ell c'_1$ in the case that e is low and in γ'_2 . So $c_2 \xrightarrow{\gamma'_1 \wedge e' \wedge \gamma'_2} c'_2$ with either $c_2 \approx_\ell c'_2$ in case e is high or $c'_2 \approx_\ell c'_1$ in case e is low. If e is high, but not a reply to the *Client*, we have $c'_1 = c_1$ or in case its a high input $c'_1 \approx_\ell c_1$. Hence when e is high, we have, $c'_2 \approx_\ell c'_1$. In the

case e is low, we have already chosen $c'_2 \approx_\ell c'_1$. We thus have $c_2||s_2 \xrightarrow{\gamma_1} c_2||s'_2 \xrightarrow{e'} c'_2||s'_2 \xrightarrow{\gamma_2} c'_2||s'_2$ with $c'_2 \approx_\ell c'_1$.

Otherwise e and e' are both replies to the *Client*. By case hypothesis, e is high and thus e' is high, since $\gamma \uparrow \ell = \langle e \rangle \uparrow \ell$.

Let δ_1 be the restriction of γ_1 to the inputs of the *Client* and δ_2 be the restriction of γ_2 to the inputs of the *Client*. By (CR2) and

(CR5) we can find c'_2 with $c'_2 \approx_\ell c'_1$ with $c_2 \xrightarrow{\delta_1 \wedge e' \wedge \delta_2} c'_2$.

We then have $c_2||s_2 \xrightarrow{\gamma_1 \wedge e \wedge \gamma_2} c'_2||s'_2$ as desired.

(b) Case: e is a low-level unblocking output to the *Client*.

We have $c_1||s_1 \xrightarrow{e} c'_1||s'_1$.

By server restrictiveness (SR2) we have $s_2 \xrightarrow{\gamma_1 \wedge e \wedge \gamma_2} s'_2$ with $s'_2 \approx_\ell s'_1$ and γ_1, γ_2 are high noninput sequences. By (SR1) and (CR1), γ_1, γ_2 do not contain unblocking outputs to the client.

Let δ_1 be the restriction of γ_1 to the inputs of the *Client* and δ_2 be the restriction of γ_2 to the inputs of the *Client*. By client restrictiveness (CR2) and (CR5), we can find c'_2 with $c'_2 \approx_\ell c'_1$ with

$c_2 \xrightarrow{\delta_1 \wedge e \wedge \delta_2} c'_2$.

We then have $c_2||s_2 \xrightarrow{\gamma_1 \wedge e \wedge \gamma_2} c'_2||s'_2$ as desired.

□

2.3.2.6 Combining multiple servers and clients

In the previous sections we composed just one client and one server. This composition can be generalized to multiple clients and multiple servers. A parallel composition of server restrictive processes is still server restrictive (using a slightly stronger version of the (SR) requirements where the replies must go to the appropriate client connection). When a client restrictive process is connected to the composite server, the resulting process is not only restrictive, but in fact server restrictive with respect to the potential

communications with other clients. So we can stepwise connect all of the clients to this process and the final system will be a restrictive system.

Note, however, that if a server needs to make blocking requests (i.e., if it also needs to play the role of the client) then a more complicated argument is needed. There are potential deadlock problems. If we can rank the servers (by some well-order, e.g., the natural numbers) so that lower ranking servers do not make calls on servers at or above their rank, then we can build a server restrictive process by inductively combining the servers. At each stage we treat the servers of the next highest rank as clients to the server built so far. The combined process will be server restrictive (as there will be no blocking reads left in the combined system).

If the servers cannot be ranked, then a composition may still be possible but one needs to use some more complex liveness property.

2.3.2.7 Discussion of condition (SR3)

None of the conditions for client and server restrictiveness, except (SR3), are very constraining. They limit some forms of nondeterministic behavior, but the conditions are unlikely to cause significant limitations on secure modeling.

However, (SR3) is fairly strong. On a high-level request from a client it must be possible to handle that request without first performing some lower level task. A simple buffering and handling of requests will not be sufficient to meet (SR3).

The (SR3) requirement is needed to prevent a small covert channel. Suppose the server needs to produce low-level outputs. If the client has a high-level input that requires a call to the server, followed by a low-level input that does not require the server, then without (SR3), the client will not be able to make outputs based on the low-level input until the server first produces its low-level outputs. So the high-level input to the client can eliminate one possible ordering of the low-level outputs. It is a rather small and hard to control channel, but it is a potential problem.

Just insisting on meeting (SR3) is not a completely satisfactory solution. The problem is that restrictiveness does not handle timing channels, yet in the end, the implementation should take steps to handle them. It is very

likely that the (unspecified) steps to solve the timing channels will sufficiently control, or eliminate, the (SR3) sequencing channel either by handling the (SR3) constraint through scheduling or by not really allowing the client to block on high-level requests. (If the timing channels are not controlled, the (SR3) channel is only a side issue by comparison.)

This difficulty is not a problem with the decomposition method presented here, but rather part of the more general problem about how much covert channel elimination should be handled at the modeling level.

If the demonstration of (SR3) is not too constraining, then we can just model server restrictiveness. Alternatives include the following:

1. One could insist on a more stringent theory than restrictiveness that would involve timing (for example, J. Millen's theory [30] or I. Sutherland's theory [57]). Indeed, such an approach does shed more insight into potential security problems. But the models will be more complicated and it may be difficult to get a reasonable characterization of the timing channels at the modeling level.
2. One could just show the model satisfies server restrictiveness without (SR3) and handle that condition during the covert channel analysis.
3. One could still use [49] in which the high-level requests do not block lower level activity.

2.3.3 Practice

It is convenient to have a simpler way of expressing and proving server restrictiveness and client restrictiveness than just using state transitions and directly showing (CR1)-(CR7) and (SR1)-(SR4). In this section, we present a modification of the method of [50] to show server restrictiveness and client restrictiveness.

The method of [50] examines models of a particular procedural syntactic form that get an input from a buffer, produce a finite number of outputs, and then wait for the next input from the buffer. The process carrying out this activity is parameterized (i.e., the process maintains information). The parameters together with the input being processed, the nondeterministic

choices, and the current location in the process syntax define the state of the process. A projection function, *proj*, on the parameters is used to describe the view of the parameters at a particular level. Security conditions are generated to check that the outputs are at the correct level and that high-level information does not leak into the lower level parameters of future transactions. For a careful presentation of this approach one should see [50].

To handle client restrictiveness we first need to extend the allowable syntactic forms to handle blocking reads. We just add a new primitive to the process specification language. (Of course the precise syntax does not matter.)

```
Request (p:port,m1:message)
  receiving (q:port,m2:message) then
    begin
      ...
      ...
    end
```

In the rest of this section we give a brief description of how to adapt the input limited restrictive methods for showing client restrictiveness and server restrictiveness.

2.3.3.1 Client

If the *Client* is represented as $Buf||P$ where *Buf* buffers all inputs except those in response to a client's requests on the server, and where *P* is the almost input limited restrictive process (i.e., input limited restrictive but with requests to the server), then $Change(Client)$ is $Buf||Change(P)$. So we can treat the request construct as if it represented the nondeterministic transitions of $Change(Client)$ and apply input limited restrictiveness analysis. Since the number of non-deterministic choices is likely to be infinite, we will need to add an extra parameter to the path analysis following each request. The level of the returning event is the level of that parameter, which must be greater than or equal to the level of the request. We will use this level definition in checking the conditions involving the levels of outputs and changes to the parameters.

If we want to include models that contain high server responses that effect the high-level part of the state, then we need to interpret the level of the extra parameter in terms of an extended projection function to make sure differences in the state will not effect future low-level outputs or the low-level parts of other parameters. If $proj$ is the original projection function describing the view of the parameters at some level, then we extend the $proj$ definition.

For any level l , old parameters x and x' , and unblocking server events y and y' :

$proj'(l, x, y) = proj'(l, x', y')$ if and only if

$(proj(l, x) = proj(l, x') \text{ and } lev(y) > l \text{ and } lev(y') > l) \text{ or } (proj(l, x) = proj(l, x') \text{ and } y = y')$.

Output and parameter conditions are generated with respect to the extended projection function.

Two states may have different parameters and so the induced equivalence relation is slightly more complicated. We must also slightly alter how states are equated to accommodate (CR4). The adjustment to handle (CR4) does not alter what output and parameter conditions should be satisfied. Also, the restrictiveness argument for $Change(Client)$ (i.e., $Buf || Change(P)$) is easily adapted.

Two states are equivalent when their input buffers are equivalent and:

1. both states are ready to handle the next transaction or are handling a high-level input, and the projection on the starting parameters are the same, or
2. both states are handling low-level requests and are “essentially” at the same point on the path, (i.e., in the process syntax), and the extended projection function on the parameters are the same.

The word “essentially” is used because we equate a state blocked after sending a high blocking request (and receiving high inputs that are not unblocking) with the next (unblocked) state.

We do not need to generate any new theorems to handle the other parts of client restrictiveness if we apply this modification to the input limited

analysis. In the next paragraph we explain why this is so.

The condition (CR1) is met by the use of the *request* construct. (It forces the reply to come immediately after the send.) (CR2) is handled as described above. (CR3) is true because one can choose either the γ' to be $\langle \rangle$ if the process is in a state performing a low-level transaction, or the γ' to finish the high-level transaction before producing the low-level output. (CR4) is true by the adjusted definition of the equivalence relation and (CR5) is true by the choice of the extended projection function. (CR6) holds because in a low-level transaction two states will be equivalent only if they are at the same spot in the process (although the high-level parts of their parameters may be different). (CR7) is true because all of the execution paths for producing outputs must be finite. (The infinite branching introduced for receiving a reply is collapsed back into one path by the introduction of the new parameter.) So we can take the rank to be the maximum number of transitions on the possible paths from a state.

2.3.3.2 Server

A process is server restrictive with respect to a list of pairs of ports, representing blocking communication with the clients. (Each client will use a pair of ports to communicate with the server.)

For each pair of ports we have to show the server condition (SR1). We can show this condition by modifying the existing output history analysis. We have to show that the output port to a client is used only if the input event is a message from that client. Further, we have to show that on any path starting with an input message from a client, exactly one reply is made back to the client using the corresponding output port. This condition can be simplified if we require the server to behave in some standard way. For example, we might require that the server transaction end with an output to the client. Special-purpose proof methods could be made available for handling models in certain standard simple forms.

To show restrictiveness (SR2) and accommodate (SR3), we need to modify the buffering method in a fashion similar to [51]. An arriving input can either temporarily preempt the current transaction or wait to be processed. The input limited restrictive part will be the same. Alternatively, if we choose

not to show (SR3), then we can just assume that a simple buffer is being used. The property (SR4) comes for free because, by the definition of the equivalence relation, both states s'_1 and s'_2 need to complete the handling of the low-level request and if s'_1 has completed handling the low-level request then s'_2 must have also completed that request. (We do not have to handle extra parameters as in client restrictiveness.)

2.3.4 Summary

The server and client restrictiveness decomposition method is sufficient to show the combined process is restrictive. It is useful because it permits blocking requests in the description of a client. It is also practical to demonstrate both of these properties with only small modifications to existing technology.

2.4 Implementation

2.4.1 Background

The Romulus toolset includes tools for establishing the restrictiveness of certain classes of processes, known as buffered server processes. Conditions sufficient for establishing state restrictiveness for this class of processes were given in [49, 50, 39]. Earlier versions of Romulus [36] implemented these conditions using a process specification language known as SL[38] and a theorem prover based on the theory of constructions [40, 37]. A later version of Romulus used a HOL based specification language and the HOL system for theorem proving [43].

This release of Romulus improves on the HOL based specification language, and improves the support for security proofs. This effort is described in the next section.

2.4.2 Server-Process Restrictiveness

This section appeared as a paper in [7].⁸

2.4.2.1 Introduction

Designing and evaluating the most trusted computer systems requires formally specifying and proving that these systems have specified security properties [12]. *Restrictiveness*, a security property developed by McCullough [21, 24, 23] is of particular interest for such an analysis. Restrictiveness is *composable*, meaning that a system composed of properly connected restrictive parts is itself restrictive.

The original version of restrictiveness, which we will call *trace restrictiveness*, was behavioral [21]; it defined system security in terms of possible sequences of input, output, and internal events. Unfortunately, proving theorems with this form of restrictiveness was extremely difficult, as it required complicated inductions over possible extensions to sequences [23, 3]. Different researchers chose two different approaches to this problem, both of which involve strengthening trace restrictiveness to provide more useful induction hypotheses:

- Alves-Foss and Levitt developed *incremental restrictiveness*, a condition on single-event extensions to event sequences; it implies trace restrictiveness and is itself composable [1, 2, 3, 4].
- McCullough [23, 25] developed what we will call *state restrictiveness*, which is usually simply called restrictiveness. State restrictiveness defines conditions on machine states that produce all of the system behavior observable at a security level, but allow no deductions about other behavior.

Rosenthal [49, 50] identified conditions sufficient to guarantee state restrictiveness in the broad case of buffered server processes. A buffered process consists of a FIFO queue and a process being buffered; it saves its inputs

⁸This section describes an older version of PSL that lacks the samepath condition and predates the development of IPSL. See Volume IV for complete details of the current implementation.

on the queue until the process being buffered is ready to receive them. The process being buffered is a server process if it waits for input in a parameterized state and processes each input by producing zero or more outputs and then calling itself to again wait for input in a possibly different parameterized state.

Sutherland, McCullough, Rosenthal, and others [36] developed process specification languages (similar to subsets of CSP [19]) for conveniently defining state machines, and they identified syntactic analyses that could be performed on such specifications to generate conditions sufficient for establishing state restrictiveness of buffered server processes. They and others at Odyssey Research Associates developed the Romulus (nee Ulysses) design analysis tool, which can take a specification of a buffered server process and compute from it a list of conditions sufficient for establishing Rosenthal’s conditions for guaranteeing state restrictiveness [36].

For largely historical reasons, the HOL version of the Romulus specification language was defined only partially, with axioms, giving extensibility but making the language and its semantics hard to understand [43]. The Romulus implementation also depended on a “meta”-level approach, generating Rosenthal-style verification conditions with an impure tactic that was itself hard to understand [43]. Further, the language’s semantics required that all state-machine parameters and all messages received as inputs or sent as outputs be coerced to a single type `:datatype`, losing the advantages of HOL’s strong typing and significantly complicating proofs of the verification conditions [43].

We developed the results presented here, using Slind’s HOL90 Release 5, to address these limitations in Romulus. We modeled our process specification language, PSL, on the Romulus specification language [43], but defined it as a concrete recursive type using Melham’s automated type definition facility [27] and gave it an operational semantics using Camilleri and Melham’s inductive definitions package [10].

We defined security properties analogous to Rosenthal-style verification conditions using the inductive definitions package, and we proved simple theorems about these properties that can be used as rewrite rules in proofs and taken as alternative definitions of the security properties, definitions requiring no extensive knowledge of security theory or HOL. Finally, we used poly-

morphic concrete recursive types to impose strong typing on state-machine parameters and message contents, in the process developing a simple technique for effectively defining processes in terms of process-valued functions.

All of our definitions are conservative extensions of the HOL logic. In addition, having the actual security definitions, abstract syntax, and operational descriptions available at the object level clarifies the intended meaning of the security properties and simplifies the proofs.

We do not intend to argue here that one version of restrictiveness is preferable to another, or that our version is equivalent to another. For the sake of simplifying the implementation, our version of restrictiveness was made intermediate in strength between trace and state restrictiveness; we believe that it could be made equivalent to Rosenthal's conditions for state restrictiveness of buffered server processes with a modest additional implementation effort.

The results presented here were part of an experiment to test the practicality of reasoning about state restrictiveness at the object level, that is, reasoning about PSL descriptions when PSL was formally embedded in HOL. These results indicate that the definitional approach is both practical and preferable.

We give an intuitive explanation of conditions giving state restrictiveness for buffered server processes in section 2.4.2.2, we formally define the process specification language PSL and give its operational semantics in section 2.4.2.3, we give security definitions in HOL using PSL in section 2.4.2.4, we specify and prove secure a simple message sorter in section 2.4.2.5, and we conclude in section 2.4.2.6.

2.4.2.2 Server-Process Restrictiveness

This section gives an informal description of conditions similar to those identified by Rosenthal [50] for showing state restrictiveness of buffered server processes.⁹ section 2.4.2.4 contains the formal definitions. The theories of trace, incremental trace, and state restrictiveness presented in [23, 3] are more general, and incremental trace restrictiveness has HOL proofs showing

⁹Restrictiveness for server processes should not be confused with "server restrictiveness," a different concept introduced by Rosenthal in [52].

that it guarantees trace restrictiveness [1, 2], HOL proofs of the sort not yet produced for our conditions. We believe, though, that our conditions are much easier to establish in the broad class of cases where they apply.

Views and Projection Functions The processes we analyze deal with multiple security levels (e.g., unclassified, confidential, secret, and top secret). We model these processes as rated state machines, which assign security levels to each input and output event. The raw information that can be obtained, or *viewed*, by an observer depends on that observer’s security level.

What an observer at a given security level can know about a system is described by three functions: functions giving the security level associated with input and output events; and a *projection function* hiding state information.

A projection function induces an equivalence partition on process states, for the process being buffered, based on security level. Two states in the same equivalence partition with respect to one level must also be in the same equivalence partition with respect to any other level dominated by that level. The projection of a state to a level determines all the information, but only the information, necessary to determine the system’s behavior that can be viewed at that level.

Restrictiveness Conditions Informally, these conditions are sufficient for guaranteeing, for buffered server processes, that an observer at a particular security level can never deduce anything about higher- or incomparable-level events:

1. Any output produced in response to an input is at a level that dominates the level of that input.
2. If the level of an input is not dominated by the observer’s level, then the projection function induces a state partition such that the process being buffered appears not to have changed state (i.e., the process’ state after responding to this input is in the same partition block as it was before receiving this input).
3. If the security level of an input is dominated by that of the observer, then any two states of the process being buffered that seemed equivalent

to the observer are not distinguished by the response to this input:

- All outputs possible for one state are also possible for the other.
- Any possible next state for one state is in the same partition block as any possible next state for the other state.
- The previous two conditions hold for any *combination* of visible outputs and seemingly equivalent next states.

The buffering guarantees that the full process is always ready to accept input, so there is never any information conveyed about the process' state by its ability or inability to accept input.

2.4.2.3 Process Specification Language

This section defines PSL, our process specification language. By “process” we mean either a state machine or a program-like description of a machine state. PSL is a simple language with the basic processes `Skip`, `Send`, `Receive`, and `Call`. `Skip` is the finished process that does nothing. `Send` transmits an output event. `Receive` takes a predicate on input events determining which input events it receives and a function determining how it responds to each event it receives. `Call` invokes the process associated with a function name and possible arguments.

Processes are combined using the PSL operators `;;`, `Orselect`, `If`, and `Buffered`. Infix operator `;;` is like the sequence operator in CSP [19]. `Orselect` is like the non-deterministic choice operator `+` in CCS [31]. `If` is the if-then-else operator. `Buffered` takes a predicate on input events, a buffer, and a process to be buffered. It returns the process that puts the input events satisfying the predicate onto the buffer, then passes them on to the process being buffered when that process is ready to receive them.

Syntax The abstract syntax of PSL is given below. It is embedded in HOL in the usual way using `define_type`. Recall that `:’name` in type signatures in HOL90 is the same as `:*name` in HOL88.

```

process =
  Skip | ;; of process # process | Orselect of process # process |
  If of bool # process # process | Send of 'outev |
  Receive of ('inev -> bool) # ('inev -> 'invoc) | Call of 'invoc |
  Buffered of ('inev -> bool) # ('inev)list # process'

```

The type variables 'inev, 'outev, and 'invoc are meant to stand for possibly polymorphic concrete recursive types of input events, output events, and function invocations. The type constructors for input and output events are meant to correspond to ports where messages enter or leave the process, and the types of the arguments to these constructors are meant to be the types of these messages; this imposes strong typing constraints on message contents. Function invocations are meant to correspond to names for calls to process-valued functions; as described in the following section on PSL semantics, they effectively allow processes to be defined in terms of process-valued functions.

Semantics There are three kinds of events: input events; output events; and silent or internal events called Tau. We introduce event types into HOL using `define_type`.

```

event = Out of 'outev | In of 'inev | Tau

```

In defining the semantics of PSL, the function *invocval* is a parameter to the function returning the transition relation that shows how PSL processes are transformed by events. The meaning of a PSL process thus varies with the mapping of invocations to PSL processes.

The 14 rules that define the behavior of PSL are shown in Figure 2.2. We use SML variable `proc` to abbreviate the polymorphic type `:('outev, 'inev, 'invoc)process`. The rules are defined inductively using `new_inductive_definition`. The function `transition` takes as a parameter a function *invocval* of type `:'invoc -> ^proc` assigning interpretations to invocations, and returns a transition relation of type `:^(proc -> ('outev, 'inev)event -> ^proc -> bool)` that shows how PSL processes of type `^proc` are transformed by events. We denote the transition from state p to state q via event e by $p \xrightarrow{e} q$. `SNOC element list` puts *element* onto the end of *list*.

$$\begin{array}{c}
\frac{}{(\text{Skip} ;; p2) \xrightarrow{\text{Tau}} p2} \qquad \frac{p1 \xrightarrow{e} px}{(p1 ;; p2) \xrightarrow{e} (px ;; p2)} \\
\\
\frac{}{(\text{Orselect } p1 \ p2) \xrightarrow{\text{Tau}} p1} \qquad \frac{}{(\text{Orselect } p1 \ p2) \xrightarrow{\text{Tau}} p2} \\
\\
\frac{b}{(\text{If } b \ p1 \ p2) \xrightarrow{\text{Tau}} p1} \qquad \frac{\neg b}{(\text{If } b \ p1 \ p2) \xrightarrow{\text{Tau}} p2} \\
\\
\frac{}{(\text{Send } outev) \xrightarrow{(\text{Out } outev)} \text{Skip}} \\
\\
\frac{}{(\text{Receive } received \ response) \xrightarrow{(\text{In } in ev)} (\text{invocval } (response \ in ev))} \\
\\
\frac{}{(\text{Call } invocation) \xrightarrow{\text{Tau}} (\text{invocval } invocation)} \\
\\
\frac{}{(\text{Buffered } buffering \ buf \ p) \xrightarrow{(\text{In } in ev)} (\text{Buffered } buffering \ (\text{SNOC } in ev \ buf) \ p)} \\
\\
\frac{p \xrightarrow{(\text{In } in ev)} px}{(\text{Buffered } buffering \ (\text{CONS } in ev \ buf) \ p) \xrightarrow{\text{Tau}} (\text{Buffered } buffering \ buf \ px)} \\
\\
\frac{p \xrightarrow{(\text{In } in ev)} px, \neg(buffering \ in ev)}{(\text{Buffered } buffering \ buf \ p) \xrightarrow{(\text{In } in ev)} (\text{Buffered } buffering \ buf \ px)} \\
\\
\frac{p \xrightarrow{(\text{Out } out ev)} px}{(\text{Buffered } buffering \ buf \ p) \xrightarrow{(\text{Out } out ev)} (\text{Buffered } buffering \ buf \ px)} \\
\\
\frac{p \xrightarrow{\text{Tau}} px}{(\text{Buffered } buffering \ buf \ p) \xrightarrow{\text{Tau}} (\text{Buffered } buffering \ buf \ px)}
\end{array}$$

Figure 2.2: PSL Process Semantics

We have also defined the semantics for the composition of processes (i.e., when the output of one process is the input of another), but for brevity omit the description. The details can be found in [8].

2.4.2.4 Security Definitions

This section defines security properties sufficient for showing security of non-parameterized PSL server processes when these processes are buffered; these security properties do not involve the projection function. Space limitations prevent giving the similarly defined properties for showing security with parameterized processes (see [8]), but the predicates `Loopsback` and `NoWritesDown` given here apply to both parameterized and non-parameterized processes.

We establish security for buffered non-parameterized server processes by examining the process *being* buffered; we do not consider the case in which the process being buffered is itself buffered. The buffered process `Buffered ($\lambda \text{inev. T}$) [] P` is secure if P is a non-parameterized server process that satisfies “all outputs are up.” A projection function need not be considered, since a non-parameterized process always starts in the same state each time it takes a new input off the buffer.

The section includes collections of theorems in if-and-only-if form that effectively define the security properties by induction on the structural complexity of PSL processes. Our tactics for showing security properties of PSL processes use these theorems as rewrite rules. These theorems can also be taken as the definitions of these properties by those unfamiliar with HOL or security theory. We derived the theorems from their implicative forms after defining the security properties with `new_inductive_definition`. We chose the definitions so that security properties hold only when they hold independently of any nondeterministic choice made and the occurrence of any input event.

A process is a server process if it receives any possible input, finishes its processing of this input, and then loops back by calling itself to wait for the next input. The predicates `Terminates`, `Loopsback`, and `BNPSP_rightform` confirm that a process is a server processes. The predicate `NoWritesDown` confirms that this process only produces “up” outputs.

Terminates *Terminates* tells whether a PSL process finishes its processing. Informally, the rules for termination are the following:

1. *Skip* always terminates.
2. $(p1 \ ; \ ; \ p2)$ terminates if both *p1* and *p2* terminate.
3. $(Orselect \ p1 \ p2)$ terminates if both *p1* and *p2* terminate.
4. $(If \ b \ p1 \ p2)$ terminates if *p1* terminates when *b* is true and *p2* terminates when *b* is false.
5. *Send* always terminates.
6. *Receive* never terminates.
7. $(Call \ invocation)$ terminates if the process invoked by $(invocval \ invocation)$ terminates.
8. *Buffered* never terminates.

The formal rules for termination follow:

$(\forall invocval. \text{Terminates } invocval \text{ Skip})$
 $(\forall invocval \ p1 \ p2.$
 $\quad \text{Terminates } invocval \ (p1 \ ; \ ; \ p2) =$
 $\quad (\text{Terminates } invocval \ p1) \wedge (\text{Terminates } invocval \ p2))$
 $(\forall invocval \ p1 \ p2.$
 $\quad \text{Terminates } invocval \ (Orselect \ p1 \ p2) =$
 $\quad (\text{Terminates } invocval \ p1) \wedge (\text{Terminates } invocval \ p2))$
 $(\forall invocval \ b \ p1 \ p2. \text{Terminates } invocval \ (If \ b \ p1 \ p2) =$
 $\quad b \Rightarrow (\text{Terminates } invocval \ p1) \mid (\text{Terminates } invocval \ p2))$
 $(\forall invocval \ outev. \text{Terminates } invocval \ (Send \ outev))$
 $(\forall invocval \text{ received response.}$
 $\quad \neg(\text{Terminates } invocval \ (Receive \text{ received response})))$
 $(\forall invocval \ invocation.$
 $\quad \text{Terminates } invocval \ (Call \ invocation) =$
 $\quad \text{Terminates } invocval \ (invocval \ invocation))$
 $(\forall invocval \ buffering \ buf \ p.$
 $\quad \neg(\text{Terminates } invocval \ (Buffered \ buffering \ buf)))$

Loopsback Loopsback tells whether a process eventually calls a named process, where the “name” is an invocation for a non-parameterized process and is an invocation type constructor for a parameterized process. Informally, the rules for looping back are the following:

1. **Skip** never loops back.
2. $(p1 ;; p2)$ loops back if $p1$ terminates and $p2$ loops back.
3. $(\text{Orselect } p1 \ p2)$ loops back if both $p1$ and $p2$ loop back.
4. $(\text{If } b \ p1 \ p2)$ loops back if $p1$ loops back when b is true and $p2$ loops back when b is false.
5. **Send** never loops back.
6. **Receive** never loops back.
7. Whether $(\text{Call } \textit{invocation})$ loops back depends on a case analysis; see the comments following this list.
8. **Buffered** never loops back.

To handle both parameterized and non-parameterized processes consistently with HOL’s typing requirements, we take the (polymorphic) type of the name of a process, $pname$, to be $: 'invoc + ('par \rightarrow 'invoc)$, where $'par$ can be instantiated with an arbitrary process-parameter type. If $pname$ names a non-parameterized process, $(\text{ISL } pname)$ is true, and if $pname$ names a parameterized process, $(\text{ISR } pname)$ is true. $(\text{Call } \textit{invocation})$ loops back, when $pname$ is a non-parameterized process, if $\textit{invocation} = (\text{OUTL } pname)$ or $(\textit{invocval } \textit{invocation})$ loops back. A similar set of conditions apply for parameterized processes. The formal rules for looping back follow:

$$\begin{aligned}
 &(\forall \textit{invocval } pname. \neg (\text{Loopsback } \textit{invocval } pname \ \text{Skip})) \\
 &(\forall \textit{invocval } pname \ p1 \ p2. \\
 &\quad \text{Loopsback } \textit{invocval } pname \ (p1 ;; p2) = \\
 &\quad (\text{Terminates } \textit{invocval } p1) \wedge (\text{Loopsback } \textit{invocval } pname \ p2)) \\
 &(\forall \textit{invocval } pname \ p1 \ p2. \\
 &\quad \text{Loopsback } \textit{invocval } pname \ (\text{Orselect } p1 \ p2) = \\
 &\quad (\text{Loopsback } \textit{invocval } pname \ p1) \wedge (\text{Loopsback } \textit{invocval } pname \ p2))
 \end{aligned}$$


```

(∀invocval pname b p1 p2.
  Loopsback invocval pname (If b p1 p2) =
  b ⇒ (Loopsback invocval pname p1) | (Loopsback invocval pname p2))
(∀invocval pname outev. ¬(Loopsback invocval pname (Send outev)))
(∀invocval pname received response.
  ¬(Loopsback invocval pname (Receive received response)))
(∀invocval pname invocation.
  (Loopsback invocval pname (Call invocation)) =
  ((ISL pname) ⇒ (invocation = (OUTL pname))) ∨
  (Loopsback invocval pname (invocval invocation))) |
  ((∃param. invocation = ((OUTR pname) param)) ∨
  (Loopsback invocval pname (invocval invocation))))
(∀invocval pname buffering buf p.
  ¬(Loopsback invocval pname (Buffered buffering buf p)))

```

BNPSP_rightform BNPSP_rightform, where BNPSP stands for “buffered, non-parameterized server process,” is true of a function mapping invocations to processes and an invocation naming a non-parameterized process if this non-parameterized process is a server process. For instantiating the predicate Loopsback in the definition of BNPSP_rightform, non-parameterized processes are treated as having the parameter (--‘one’--). For brevity, we do not give receivesall and reaction here, but they are straightforward. The predicate receivesall is true if it is applied to a PSL Receive process which receives arbitrary input events. The conversion rule reaction is applied to an input event, a function mapping invocations to PSL processes, and a PSL process. If this process is a Receive, reaction returns the process that the Receive process changes into in response to the input event.

```

(∀invocval nppname.
  BNPSP_rightform invocval nppname =
  receivesall (invocval nppname) ∧
  (∀inev.
    Loopsback
      invocval (INL nppname) (reaction inev invocval (invocval nppname))))

```

NoWritesDown **NoWritesDown** is defined for a dominance relation, a level-assignment function for outputs, a security level assumed to be the level of some input, a function mapping invocations to processes, a process name, and a PSL process. It holds if, before the process ends with a call to the named process, the level of every output produced dominates the input level. Informally, the rules for always producing “up” outputs are the following:

1. Skip satisfies **NoWritesDown**.
2. ($p1 \ ; \ ; \ p2$) satisfies **NoWritesDown** if both $p1$ and $p2$ satisfy it.
3. (**Orselect** $p1 \ p2$) satisfies **NoWritesDown** if both $p1$ and $p2$ satisfy it.
4. (**If** $b \ p1 \ p2$) satisfies **NoWritesDown** if $p1$ satisfies **NoWritesDown** when b is true and $p2$ satisfies **NoWritesDown** when b is false.
5. (**Send** $outev$) satisfies **NoWritesDown** if the level of $outev$ dominates the level of the input.
6. Receive never satisfies **NoWritesDown**.
7. (**Call** $invocation$) satisfies **NoWritesDown** if the invocation is a call to $pname$ or if ($invocval \ invocation$) satisfies **NoWritesDown**. The cases are similar to those for **Loopsback**.
8. Buffered processes never satisfy **NoWritesDown**.

The formal rules defining **NoWritesDown** follow:

$$\begin{aligned}
 &(\forall \text{dom outlev level invocval pname.} \\
 &\quad \text{NoWritesDown dom outlev level invocval pname Skip}) \\
 &(\forall \text{dom outlev level invocval pname } p1 \ p2. \\
 &\quad \text{NoWritesDown dom outlev level invocval pname } (p1 \ ; \ ; \ p2) = \\
 &\quad (\text{NoWritesDown dom outlev level invocval pname } p1) \wedge \\
 &\quad (\text{NoWritesDown dom outlev level invocval pname } p2)) \\
 &(\forall \text{dom outlev level invocval pname } p1 \ p2. \\
 &\quad \text{NoWritesDown dom outlev level invocval pname } (\text{Orselect } p1 \ p2) = \\
 &\quad (\text{NoWritesDown dom outlev level invocval pname } p1) \wedge \\
 &\quad (\text{NoWritesDown dom outlev level invocval pname } p2))
 \end{aligned}$$

$(\forall \text{dom outlev level invocval pname } b \text{ } p1 \text{ } p2.$
 $\text{NoWritesDown dom outlev level invocval pname (If } b \text{ } p1 \text{ } p2) =$
 $b \Rightarrow (\text{NoWritesDown dom outlev level invocval pname } p1) \mid$
 $(\text{NoWritesDown dom outlev level invocval pname } p2))$
 $(\forall \text{dom outlev level invocval pname outev.}$
 $\text{NoWritesDown dom outlev level invocval pname (Send outev) =}$
 $(\text{dom (outlev outev) level}))$
 $(\forall \text{dom outlev level invocval pname received response.}$
 $\neg(\text{NoWritesDown}$
 $\text{dom outlev level invocval pname (Receive received response))$
 $(\forall \text{dom outlev level invocval pname invocation.}$
 $\text{NoWritesDown dom outlev level invocval pname (Call invocation) =}$
 $((\text{ISL pname}) \Rightarrow$
 $((\text{invocation} = \text{OUTL pname}) \vee$
 $(\text{NoWritesDown dom outlev level invocval pname (invocval invocation))) \mid$
 $((\exists \text{param. invocation} = ((\text{OUTR pname}) \text{ param})) \vee$
 $(\text{NoWritesDown dom outlev level invocval pname (invocval invocation))))$
 $(\forall \text{dom outlev level invocval pname buffering buf p.}$
 $\neg(\text{NoWritesDown}$
 $\text{dom outlev level invocval pname (Buffered buffering buf p))$

BNPSP_restrictive **BNPSP_restrictive** is a relation between a dominance relation on security levels, a function mapping input events to security levels, a function mapping output events to security levels, a function mapping invocations to processes, and an atomic invocation.

$(\forall \text{dom inlev outlev invocval nppname.}$
 $\text{BNPSP_restrictive dom inlev outlev invocval nppname1} =$
 $(\text{BNPSP_rightform invocval nppname}) \wedge$
 $(\text{NoWritesDown dom inlev outlev invocval nppname}))$

2.4.2.5 Example

This section presents an example using our techniques to specify a process and prove it secure. The example is a sorter for a token ring station. It sends out signals showing the receipt of the token or a message from the host station and sends on messages from other stations.

We first define input events, output events, sorter invocations, and sorter PSL processes. For generality, we let the types of token ring stations and message contents be given by type variables. There are two invocations, one for the sorter itself and the other for giving the sorter's response to input events.

```

val SortInEv_Def =
  define_type{name = "SortInEv_Def",
    type_spec =
      'SortInEv = All of bool # 'station # 'station # 'data',
      fixities = [Prefix]};
val SortInEv = ty_antiq(==:('station,'data)SortInEv'==);
val SortOutEv_Def =
  define_type{name = "SortOutEv_Def",
    type_spec =
      'SortOutEv = Others of bool # 'station # 'station # 'data |
        Host of one | Tokens of one',
      fixities = [Prefix,Prefix,Prefix]};
val SortOutEv = ty_antiq(==:('station,'data)SortOutEv'==);
val SortInvoc_Def =
  define_type{name = "SortInvoc_Def",
    type_spec = 'SortInvoc = Sorter | SortInput of ^SortInEv',
    fixities = [Prefix,Prefix]};
val SortInvoc = ty_antiq(==:('station,'data)SortInvoc'==);
val SortProc =
  ty_antiq(==:(^SortOutEv,^SortInEv,^SortInvoc) process'==);

```

Now we introduce a constant for an unspecified function assigning security levels to token ring stations, a constant for the host station, and a constant for the lowest security level. Again for generality, we let the type of security levels be given by a type variable. With these, we define the functions assigning security levels to input and output events. For input events, the token is assigned `systemlow` and other messages are assigned `station_level` of the station that sent them. For output events, messages passed on from another station are assigned the level of the message's sender, signals showing receipt of a message from the current station are assigned the level of the current station, and signals showing receipt of the token are assigned `systemlow`.

```

new_constant{Name="station_level",Ty==:'station->'level'==};
new_constant{Name="this_station",Ty==:'station'==};
new_constant{Name="systemlow",Ty==:'level'==};

```

```

new_recursive_definition{name="SortInLevel",
  fixity=Prefix, rec_axiom=SortInEv_Def,
  def= let val SortInLevel= --'SortInLevel:~SortInEv->'level'-- in
    --'(^SortInLevel (All tokenflag sender receiver data) =
      (tokenflag => systemlow | (station_level sender)))'-- end};
new_recursive_definition{name="SortOutLevel",
  fixity=Prefix, rec_axiom=SortOutEv_Def,
  def= let val SortOutLevel= --'SortOutLevel:~SortOutEv->'level'--;
    val station_level= --'station_level:'station->'level'--; in
    --'(^SortOutLevel (Others tokenflag sender receiver data) =
      (^station_level sender)) /\
      (^SortOutLevel (Host x) = (^station_level this_station)) /\
      (^SortOutLevel (Tokens x) = systemlow)'-- end};

```

We define the dominance relation on security levels as the reflexive-transitive closure of an unspecified basic order on them, define the process-valued functions interpreting the invocations, and define the function mapping the invocations to their interpretations. The process sorter, interpreting Sorter, waits for an arbitrary input event and invokes SortInput on that event. The function sortInput, interpreting SortInput, sends event (Tokens (--'one'--)) after receiving the token, sends event (Host (--'one'--)) after receiving a message from the host station, sends an event passing on the received message otherwise, and then calls the sorter process to wait for the next input.

```

new_constant{Name="basic_order",Ty= ==': 'level->'level->bool'==};
new_definition("dom",
  let val dom = --'dom:'level -> 'level -> bool'-- in
    --'(^dom x y) = RTC basic_order x y'-- end);
new_definition("sorter",
  --'sorter:~SortProc = (Receive (\ev:~SortInEv. T) SortInput)'--);
new_recursive_definition{name="sortInput",
  fixity=Prefix, rec_axiom=SortInEv_Def,
  def =
    let val sortInput = --'sortInput:~SortInEv -> ~SortProc'-- in
      --'(^sortInput (All tokenflag sender receiver data) =
        (If tokenflag
          (Send (Tokens one))
          (If (sender = this_station)
            (Send (Host one))
            (Send (Others tokenflag sender receiver data)))))) ;;
      (Call Sorter))'-- end};

```

```

new_recursive_definition{name="SortInvocval",
  fixity=Prefix, rec_axiom=SortInvoc_Def,
  def =
    let val SortInvocVal= --'SortInvocVal:~SortInvoc->~SortProc'--in
      --'(^SortInvocVal Sorter = sorter) /\
        (^SortInvocVal (SortInput ineq) = (sortInput ineq))'-- end};

```

Now we give the proof that the sorter is secure, omitting the trivial proof that the dominance relation on levels is transitive as it was defined to be. For brevity, we do not give our specialized tactics here (see [8]), but they are straightforward; they expand out definitions, do case splits on possible input events, and apply structural-complexity rewrite rules until *all PSL constructs and all security predicates defined in terms of them disappear*. The only subgoals not proved automatically are “all outputs are up” conditions that all follow from the reflexivity of the dom relation.

```

val BNPS_restrictive =
  --'BNPS_restrictive:
    ('level ->'level->bool) -> (^SortInEv->'level) ->
    (^SortOutEv->'level) -> (^SortInvoc->~SortProc) -->
    ^SortInvoc -> bool'--;
g('BNPS_restrictive
  dom SortInLevel SortOutLevel SortInvocVal Sorter');
use "/home/projects/romulus/chin/new/romtactics.sml";
add_definitions_to_sml "-";
e(BNPS_restrictive_TAC THEN
  ASM_REWRITE_TAC [SortOutLevel, SortInLevel] THEN
  MATCH_ACCEPT_TAC dom_reflexive);

```

The three-line proof here is as simple as the intuitive reason why the sorter is secure: Every output has the same level as the input that causes it. In a parameterized-process example comparing our techniques to the earlier approach of using type coercions and impure tactics, a “high water mark” file system, our techniques produced a clearer, more securely founded proof that was shorter by roughly a factor of four [8].

2.4.2.6 Conclusions

We have shown that making HOL object-language definitions of a process specification language and security properties sufficient for guaranteeing se-

curity of buffered server processes is both possible and practical, and seems preferable to an earlier, meta-level approach. In doing so, we have developed techniques for imposing strong typing restrictions on security specifications and effectively defining processes in terms of process-valued functions.

Possible future work includes strengthening our security conditions to make them equivalent to the conditions identified by Rosenthal [50] for establishing state restrictiveness, and then constructing a HOL proof that these conditions guarantee state restrictiveness for buffered server processes.

Chapter 3

Theories of Integrity

In this chapter we present the formal theories of integrity used in Romulus. Computer system integrity is a catch all phrase to describe the assurance that data is protected from unauthorized modification or destruction. (For a general discussion of integrity see [33].) Our discussion of integrity is divided into two general areas. In section 3.1 we consider some selected models of integrity. In section 3.2 we consider authentication protocols, a standard technique for establishing the correctness of authorization.

3.1 Integrity Models

In this section we present a collection of formal properties that are meant to formalize various notions of *integrity* for computer systems. The word “integrity” is used to mean many things in the software community, from very specific meanings like consistency of replicated data in a distributed database to very general meanings amounting to “correct functioning”. (For a general discussion of integrity see [33].) We have not tried to cover all possible meanings of the word “integrity”. We have focussed on (1) formalizing some of the meanings of integrity that are currently used in the computer security community, such as the Biba model and the Clark-Wilson model, and (2) doing some original work on meanings of “integrity” that are relevant to current concerns in the distributed computing community, such as distributed

data integrity and distributed authentication.

In section 3.1.1 we give some simple requirements on theories of integrity, including some informal meanings and some threats to integrity that we want our theories to capture. In section 3.1.2 we describe some techniques for ensuring some of the meanings of integrity that we want our formal theories to deal with. In section 3.1.3 we describe the way we will formally represent systems with integrity requirements as mathematical objects. In section 3.1.4 we present the formal properties we have developed to capture various aspects of integrity. In section 3.1.5, we give a semiformal example we have formulated to help drive the development of our formal theories. In Appendix A we give some technical details of a construction associated with one of our integrity theories.

3.1.1 Requirements of Integrity Theories

In order for our theories of integrity to be reasonable, they must meet two requirements. First, they must capture certain informal notions of integrity, in the sense that systems that do not meet those informal notions of integrity should fail to meet one of our theories, and conversely, that systems that satisfy those informal notions of integrity should satisfy one of our theories. Second, our theories of integrity must capture certain threats to integrity, in the sense that systems which do not include adequate countermeasures to those threats should fail to satisfy one of our theories.

The informal meanings of integrity that we mean to capture with our theories include the following:

- Data is trustworthy, that is, the data that users get from the system is reliable.
- Users cannot spoof the system.
- Data cannot be corrupted, i.e. modified in inappropriate ways.
- Certain operations are accessible only to authorized users.

The threats to integrity we mean to capture with our theories include the following:

- Distributed information management and concurrency control problems: concurrent updates, concurrent access, interleaving of atomic parts of nonatomic transactions.
- Trojan horses and viruses
- Modification in unprotected media (for example, transmission media)
- Inadequate authentication

3.1.2 Techniques for Ensuring Integrity

In this section we describe some of the techniques for ensuring integrity that we want our formal theories to deal with, in the sense that some of our formal theories may be satisfied by the proper use of these techniques.

3.1.2.1 Locks and Protocols

First of all, we wish to be able to deal with some of the techniques that are used to ensure consistent management of distributed data. Specifically, we wish to be able to represent and prove systems that use *data locks* and *agreement protocols*. A data lock is a simple primitive for controlling the order in which accesses are made to a piece of data. At any given time, a piece of data is either locked or unlocked. A process may attempt to lock a piece of data at any time. If the data is currently not locked, it becomes locked, and the process proceeds. If the data is currently locked, the process is suspended pending unlocking. When a piece of data is unlocked, if there are any processes waiting to lock it, one of them is chosen according to some algorithm, and that process locks the data and proceeds. Locks on data can be used to ensure that all the atomic actions associated with a given transaction occur and finish before those associated with another transaction start.

Agreement protocols are another technique for ensuring consistent access to distributed data. Agreement protocols are protocols which exchange messages between remote sites to ensure that the various sites agree on a consistent picture of the distributed data in the system. We will see a simple example of this in section 3.1.5.

3.1.2.2 Type Enforcement Mechanisms

Type enforcement mechanisms can be used to ensure conditions like those in the Clark-Wilson model, for example, that certain kinds of data objects can only be accessed by certain operations. By making those data objects elements of a special type, and making the trusted operations on them the sole operations on that type, ordinary type-checking will enforce the integrity requirements.

3.1.2.3 Cryptographic Techniques

Another group of techniques we wish to address with our theories is techniques based on the use of cryptography. For readers who are not familiar with these techniques, we include the following short tutorial on cryptographic techniques for ensuring integrity.

Information processing and communication environments are constantly under threat of adversaries who attempt to eavesdrop (passive attacks) and tamper (active attacks) with information in storage and in transit. In passive attacks, adversaries try to deduce sensitive information by observation, while in active attacks, they try to modify or forge information.

Unless in an isolated environment where physical protection against such attacks is possible and available, system integrity (including information secrecy and integrity) is best protected by cryptographic measures, such as encryption and digital signature, which use cryptographic algorithms together with keys. The success of these cryptographic measures relies solely on the keys being kept secret so malicious parties cannot obtain the keys by legitimate or illegitimate means.

This section gives a brief introduction to the cryptographic methods for assuring integrity. First, the methods themselves are introduced. We then examine how they are used to provide information secrecy and integrity. After that, authentication and key distribution, which are integral parts of the application of cryptographic methods, are introduced, pending a further discussion in section 3.2.

Cryptographic Methods in a Nutshell The basic cryptographic mechanisms are *one-way functions* (including *one-way hash functions*) and *cryptosystems* (both conventional cryptosystems and public-key cryptosystems). They are the basis on which more complicated cryptographic mechanisms are built. We will not specifically discuss these more complicated mechanisms, because they are all built out of one-way functions and cryptosystems.

One-Way (Hash) Functions A one-way function is also known as a modification detection code [29], a fingerprint [46], a one-way cipher [62], a message digest algorithm [47], and by other terms. Needham first introduced the idea of a one-way cipher in a login procedure for the Cambridge multiple-access system [62]. Diffie and Hellman gave a more formal definition of a one-way function [13]. Merkle gave the first definition of a one-way hash function. He also made a distinction between a weak and a strong one-way hash function [28]. For brevity, we only introduce strong one-way hash functions, since their range of application is the widest. Note that the role of hashing is the traditional one — to reduce the size of the data (in this case, the data to be encrypted).

A hash function f is a strong one-way function if (1) given $f(x)$, it is computationally infeasible to compute x ; (2) given x and $f(x)$, it is infeasible to compute y such that $x \neq y$ but $f(x) = f(y)$; (3) it is infeasible to compute a pair of inputs x and y such that $x \neq y$ but $f(x) = f(y)$.

Cryptosystems A cryptosystem is also commonly referred to as an encryption system. It includes an encryption algorithm and a decryption algorithm. It can be used in various ways, including as a one-way hash function.

An encryption algorithm E represents a family of functions. An encryption key k uniquely determines a particular member of the family (except perhaps in the case of randomized encryption, which will not be addressed here), denoted as $E(k, \cdot)$. There is another corresponding decryption D , which is also a family of functions. For a key, the uniquely determined decryption function is the inverse of the encryption function that is determined by the same key, i.e. $D(k, E(k, \cdot)) = I$, where I is the identical function. In other words, for any x and k , $D(k, E(k, x)) = x$.

E has the property that given $\{(x_i, E(k, x_i) \mid i = 1, \dots, n\}$, $E(k, x)$ and y , it should be impossible or computationally infeasible for any party who does not possess k to compute x or $E(k, y)$.

The encryption algorithms existing today fall into two categories [13]. One is called the conventional encryption algorithm, typified by the Data Encryption Standard (DES) [54]. Here, the encryption key is the same as the decryption key, so the originator and recipient of an encrypted message must share the same key. The second one is called public-key algorithms. Here, each party has a pair of corresponding keys, a public key and a private key. The public key can be known to anyone while the private key is never published and is known only to the owner. An originator will use the recipient's public key to encrypt a message and the recipient will use its private key to decrypt it. The most widely studied and used public-key algorithm is the RSA scheme [48]. The RSA scheme also has a property that if an originator encrypts a message with its private key, then anyone can verify the identity of the originator by using the corresponding public key to decrypt the message. This particular usage constitutes a digital signature scheme, the encrypted (or signed) message being the signature. Although digital signature schemes are usually based on public-key systems, they could also be built on top of conventional systems.

A public-key system offers some advantages over a conventional system. For example, suppose that two communicating parties are far apart physically and only conventional encryption is used. If one party is compromised, all secret information will be assumed to be known to malicious parties, so it is necessary to immediately update the key the two parties have been using. If the other party is on a satellite, the update would not be as easy as asking a user to go to the administrator's office and choose a new password. If public-key technology is used, it is merely necessary to inform the satellite (securely) that someone's public key has been compromised. It is never necessary to change the satellite's public key because no one else ever knows its private key. Thus damage is confined locally.

To use the public-key technology, several mechanisms must be available. One is the generation of quality public and private key pairs. Another is a registration of public keys maintained by a trusted authority. However, it is widely known that public-key encryption algorithms are generally slower and

more costly than conventional encryption algorithms. For example, DES is much more efficient than RSA in routinely encrypting large amounts of data. Fortunately, it is feasible for the parties to use their public-key technology to first establish a session key for each conversation session, using a Needham-Schroeder style protocol, and then use the session key to encrypt subsequent communication. Such a hybrid scheme not only combines the best of the two types of encryption schemes but also has an added security, because now each session uses a different key and the public keys are used much less often, so it is much harder to break them.

A basic encryption algorithm works on a certain length of input (*plaintext*). For example, DES operates on a block of 64 bits. It is not difficult to extend the basic algorithm to handle longer plaintexts. For example, DES can be used in a CBC mode [54] which uses a chaining method to encrypt arbitrarily long plaintexts. A block in RSA is typically around 512 bits, and a similar chaining method can be used there too.

Assuring Information Secrecy Given an encryption algorithm and a key, a plaintext x can be transformed into a piece of ciphertext $E(k, x)$. Given the ciphertext, the intended recipient who possesses the relevant secret information (for example, the key) can perform a decryption on the ciphertext to retrieve the plaintext, since $D(k, E(k, x)) = x$.

According to E 's property, given a number of corresponding pairs of plaintexts and ciphertexts, and a particular ciphertext $E(k, x)$, it should be impossible or computationally infeasible for any party who does not possess the key k to compute the corresponding plaintext x . Thus encryption helps to protect the secrecy of the plaintext. An encryption algorithm with this property is resistant to so-called known-plaintext attacks.

There are more types of attack against which the strength of an encryption can be evaluated. The weakest is probably the ciphertext-only attack. A stronger one is the chosen-plaintext attack. A good encryption algorithm should be resistant to all such attacks. For example, DES and RSA have been extensively subjected to such attacks and no significant weakness has been reported. Note that generally the longer the key, the more difficult it is to break a cryptosystem. DES's key length is 64 bits, which is regarded as sufficient for some applications and insufficient for others. RSA is typically

used with keys of around 512 bits, though there is no theoretical limit on the key length.

Assuring Information Integrity Techniques for authenticating texts have existed in the literature for many years. Traditionally, these methods have depended upon encryption. Recall the property of encryption that without knowing key k , it is infeasible to compute $E(k, y)$ for an arbitrary y . Therefore, if the ciphertext is modified in any way, decryption with the same key will not recover the original plaintext. Most probably, such a modification will destroy the structure of the plaintext, so that decryption will produce an illegible text. Thus any modifications to the ciphertext can be detected, and integrity of the plaintext is provided in the sense that forgery is always detectable.

Integrity can also be protected by other means. For example, for any text x , one could compute a checksum $f(x)$, where f is a one-way (hash) function. Now anyone who possesses x' and $f(x)$ can easily verify whether x' is a copy of x or not. Note here that of course the verifier has to compare with the original checksum, so the integrity of the checksum needs to be protected, by encryption perhaps. Now if f is also a hash function, one needs only to encrypt the checksum, which is significantly shorter than the original text x .

Methods for authenticating texts based upon *pseudorandom functions* are also useful. Informally, a pseudorandom function f has the property that if f is unknown, it is computationally infeasible to produce $f(x)$ for any x with a probability of success greater than random guessing, even after having seen several other $\langle x', f(x') \rangle$ pairs. Thus, given a family of pseudorandom functions $\{f_k\}_{k \in \mathcal{K}}$, indexed by keys from some key space \mathcal{K} , one who possesses the secret key k can compute $f_k(x)$. Later one can authenticate text x by recomputing this function and compare the result with the earlier value [45]. In practice, this use of pseudorandom functions is usually approximated by defining $f_k(x) = g(k, x)$, where g is a one-way hash function, because several efficient implementations of one-way hash functions exist (for example, [47]).

Authentication and Key Distribution Authentication and key distribution is the subject of a major Romulus effort, which is described in section 3.2.

Some preliminary arrangement has to be made before encryption or signature signing can begin. In the case of conventional encryption, the parties wishing to communicate must first establish a shared secret key among themselves. In the case of public-key encryption, each party has to choose its pair of keys and register the public key at a trusted place, for example, a key directory managed by a trusted authority. This phase of the operation is generally done via a key distribution protocol, sometimes also known as an authentication protocol.

The process to arrange this is called key distribution. Needham and Schroeder described the principles for key distribution [34]. The basic idea is to establish a trusted third party — an authentication server or key distribution server — whose responsibility includes initial key assignment and subsequent key changes.

Initially, each user must acquire a key in a secure fashion. In particular, a user can choose or be assigned a suitable password as in a typical system. Alternatively, a user can use a so-called “smart card”. It is a card of the size of a credit card, which stores important information, some of which is difficult to remember for a human user, such as a very long key. The card can only be used in conjunction with a Personal Identification Number (PIN). Each system component also securely receives a key, for example, at booting time, to communicate with the key distribution server. This initial set up probably involves human intervention. Afterwards, parties who wish to communicate with each other must ask the server to set up session keys for them through a key distribution protocol or an authentication protocol [5, 11, 34, 35, 44, 55].

The advantages of this arrangement are manifold. The server is trusted to generate and distribute quality keys so that cryptographic attacks on weak keys are less likely to succeed. Session keys are changed frequently so that other forms of cryptographic attack, such as known-plaintext attacks and dictionary attacks, are impossible or very difficult. Each party does not need much secure storage to communicate to a large number of partners.

3.1.3 System Representations for Integrity

3.1.3.1 State Machine Representation

The way we will represent systems with critical integrity requirements is based on the representations of systems used in the restrictiveness model, namely, state machines that interact with their environment by exchanging events. The relevant definitions are given in section 2.1.2.1 and section 2.1.3.

3.1.3.2 Representing Probabilistic Properties

The principal extension we will make to the state machine framework for purposes of integrity analysis is to incorporate probabilistic information about the machine's execution into the model. This is relevant to integrity for two reasons. First, we need probabilistic information in order to do a nontrivial analysis of systems which use encryption (see section 3.1.4.4 for more detail). Second, we eventually want to be able to analyze the reliability of data from a probabilistic point of view, for example, determining probabilistic degrees of confidence in data.

The simplest way to incorporate probability would be to attach probabilities (real numbers between 0 and 1) to state transitions, with the caveat that for each state s , the sum of the probabilities for all the transitions from s to another state, accompanied by some event, is 1. We will call such an assignment a *probabilistic state transition relation* (or just "probabilistic transition relation" for short). In fact, this is almost what we will do, with a slight modification.

The problem with the above simple approach is that it requires that we determine probabilities for *all* state transitions, including state transitions accompanied by inputs. The probabilities of inputs, however, are not determined by the system being designed or analyzed, but by the environment. An assignment of probabilities to input transitions should not be part of a system's design.

We can fix this problem in the following simple way: instead of defining a single probabilistic transition relation, we define a set S of such relations, and specify that the probabilistic transition relation that the system actu-

ally obeys is one of those in S . In the ideal case, the set S will place no restrictions on the probabilities of input transitions. In other words, for every assignment A of probabilities to input transitions, there will a probabilistic transition relation $R \in S$ which agrees with A . In practice, however, we may need to specify certain restrictions on how the environment behaves. These restrictions can be expressed by choosing the set S so that only certain assignments of probabilities to input transitions occur in S .

To illustrate how the set S might be defined, we will give a simple example. We will first describe the example informally, and then give a mathematical description in English. The system we have in mind is a system which has two inputs, “0” and “1”. When the system receives an input, it stores the value input. At any given time, the system can probabilistically choose either to emit the value it has stored, or 1 – the stored value. We suppose that it chooses to output the value stored r times more frequently than it chooses to output the opposite.

We can represent this system by a state machine with 3 states: “0”, “1”, and “null” (“null” being the state in which the system has no last input to remember, either because none has occurred yet or because it has made an output). The initial state is “null”. The set S is the set of all probabilistic transition relations T such that:

1. T assigns probability 0 to any transition where the accompanying event is an input i (0 or 1) and whose final state is not the same as i . (In other words, input transitions must change the state to remember what was input).
2. T assigns probability 0 to any internal transition which is not a null transition. (In other words, the state can only change by external transitions).
3. T assigns probability 0 to any output transition whose first state is “null”. (In other words, outputs can only happen from a state in which the machine is remembering some previous input value).
4. T assigns probability 0 to any output transition whose final state is *not* “null”. (In other words, outputs clear the value being remembered by the machine).

5. The probability T assigns to an output transition from a state $s \neq \text{null}$ to state “null” with output s is r times the probability T assigns to the same transition with output $1 - s$.

This set of probabilistic transition relations captures the informal specification of the machine, including the relative probability of emitting the opposite of the value that was input, but does not bias the probabilities of various input events. There are T 's in S in which nothing but 0's are input, where nothing but 1's are input, where 0's and 1's are input according to some distribution, etc.

For a given $T \in S$, we can define the probability of a given finite trace of the machine. This probability is just the product of the probabilities of each of the transitions in the finite trace. This can in turn be used to define a *probability measure* for the complete traces. A probability measure is a function which takes certain sets of complete traces and returns the probability of a complete trace being in that set. Since the definition of this probability measure is a bit technical, we will defer it to an appendix. Proofs of probabilistic properties of a probabilistic state machine are essentially proofs that for every $T \in S$, a certain property holds of the probability measure generated from T . The definition of a probability measure, and a number of classical results about probability measures, can be found in [53]. We will discuss probability measures again in section 3.1.4.4 when we define probabilistic inference.

3.1.4 Formal Integrity Properties

In this section we describe the formal theories we have developed to express our informal notions of integrity. In section 3.1.4.1 we describe a simple version of restrictiveness with integrity levels instead of security levels. This model stands in relation to restrictiveness as the Biba integrity model stands to the Bell-LaPadula model of security. In section 3.1.4.2, we describe a formal theory of Clark-Wilson-like requirements that certain data only be accessed by certain operations, and that certain operations only be invoked by certain users. In section 3.1.4.4, we describe a probabilistic version of deducibility security which is a first step towards being able to analyze critical systems which use encryption.

3.1.4.1 Restrictiveness with Integrity Levels

The Biba integrity model is the dual of Bell-LaPadula, that is, it can be gotten by taking the Bell-LaPadula model and (1) replacing security levels with integrity levels, and (2) turning the order relation around in the simple and *-properties. Similarly, we can take the dual of restrictiveness by replacing security levels with integrity levels and turning the order relation around, that is, replacing “high” with “low” everywhere and making the low security projection into the high integrity projection. This makes a Biba-like integrity policy which requires that low integrity information cannot flow into high integrity information, even through covert channels. Like Biba, it has the drawback that, unlike security levels, there are no standard integrity classifications or categories which experience has shown captures important degrees of integrity or trustworthiness. However, given the more general paradigm for representing information in restrictiveness, we could use integrity categories to control authorization.¹ For example, if only certain users were allowed to modify certain files, use certain resources, or use certain devices, the various privileges could be equated with integrity categories, and restrictiveness for integrity would require that individuals who do not have the privilege (category) would not be allowed to affect the appropriate objects. We generalize this use of restrictiveness to limit authorization in the next section.

3.1.4.2 Authorization Requirements Stated as Restrictiveness

Many integrity requirements in Clark-Wilson are requirements that various activities can only take place if authorized, and can only take place via a certain limited interface (trusted operations). There are also “metaauthorization” requirements that certain authorization is required to change authorization.

We can formalize some requirements of this sort by an application of a form of restrictiveness. Suppose a certain system has a requirement that a certain data item D can only be accessed by a certain operation O . We formalize this requirement in the following steps. First, we define the values that D can have. Next, we define a notion of the “state” of an invocation

¹In fact, the categories of the XTS-2000 are being used for this very purpose in the design of the WWMCCS/CAT guard for MAC

of O . This state will include things like the parameters to the invocation, internal variables of O , any other parts of the system state which O may affect, etc. Next, we define how O acts on D as a state machine (which we will call the O/D state machine) whose states are (1) a value for D , and (2) a collection of states of invocations of O . The O/D machine will have its own transition relation which will define how invocations of O act on D . The O/D machine may have events of the system state machine as part of it. Finally, we define a projection function *proj* which takes a state of the system and returns a state of the O/D state machine. We treat this projection function as the “low” projection in restrictiveness, and we call an input or output “low” only if it is an event of the O/D state machine. All other inputs and outputs are “high”.

We can now define a second transition relation on the O/D machine, which it “inherits” from the system state machine. This transition function is defined as follows: a transition from state s of the O/D machine to state s' of the O/D machine with event e is inherited from the system machine if and only if there exist states t and t' of the system state machine and an event e' such that:

1. the system machine can make a transition from t to t' accompanied by event e'
2. $proj(t) = s$ and $proj(t') = s'$
3. either (1) $e = e'$ or (2) e' is not an event of the O/D machine and e is the silent event.

We formalize the requirement that all modifications of D are performed by O as: the transition relation the O/D machine inherits from the system state machine is the same as the state transition of the O/D machine possibly with some extra null transitions added.

This formal statement captures the authorization requirement because every change in the value of D that can be made by the system must correspond to some behavior of the O/D state machine, and the O/D state machine represents only the actions of invocations of O . Thus, all modifications of D by the system must correspond to modifications made by an invocation of O .

This general approach can also be used to formalize requirements that invocations of certain operations O can only be initiated through certain interfaces I (for example, the system manager's interface, or the system security officer's interface). This can be done by associating I with the set of events that correspond to inputs to and outputs from I . We then define a state machine (which we will call the O/I machine) for that interface whose state includes a set of states of invocations of O , and whose events are exactly those associated with I and O . We then define a projection function which takes a state of the system machine and returns a state of the O/I machine. As in the case of the O/D machine, we define a transition relation that the O/I machine inherits from the system machine, and require that the inherited transition relation be the same as the O/I transition relation except for null transitions. As in the case of the O/D machine, this formal statement requires that all invocations of O in the system must correspond to invocations that take place in the O/I machine alone, and so must correspond to operations initiated through the I interface.

Formal requirements of the above sort are the first step towards a formal theory of the sort of integrity dealt with by the Clark-Wilson model. In particular, most of the so-called "enforcement rules" of the model are of one of the above forms. We have not tested formal requirements of the above types in examples yet. We are, however, currently formulating an operating system model which we will apply these requirements to.

3.1.4.3 Belief Models of Integrity

In some sense, integrity levels, integrity categories, trusted interfaces to data, and authorization rules are all trying to accomplish a more high-level goal, namely, the goal of ensuring that the data contained in the system is an accurate reflection of reality, whether it is the reality of the system itself or of its environment. Integrity levels attempt to keep "high integrity" data (which presumably is a very good reflection of reality) from being "tainted" by "low integrity" data. Authorization rules attempt to keep data "untainted" by the manipulations of individuals who cannot be trusted to maintain its correspondence with reality. Trusted interfaces attempt to ensure that no one, trustworthy or not, can make certain accesses to data which might damage its reliability.

In this section we describe a general approach to formalizing the high-level goals of such measures directly, that is, integrity requirements of the form:

“The system only tells you true information.”

or, to put it another way,

“The beliefs you hold based on what the system has told you are always true.”

We will begin by formalizing some of the notions in terms of which the informal definition is stated.

What is meant by a “belief” that is held by some entity within or external to the system? A user of an information system might believe things like:

1. that a certain variable, file, or field of a database has a certain value v
2. that some external entity e made certain inputs i at some point in the past
3. that some other entity has received certain outputs o

All of these beliefs can be expressed in the form “the user believes that the finite trace he is in is in a certain set S .” In case 1 above, S is the set of all finite traces t such that the variable, file, or field has value v at the end of t (i.e. in the state at the end of t). In case 2 above, S is the set of all finite traces t such that e has made inputs i in the course of t . In case 3 above, S is the set of all finite traces t such that outputs o have occurred in t . We will therefore formalize beliefs as sets of finite traces.

What does it mean for the system to “tell” an entity something? First of all, the “something” told will be a belief, as defined above. Thus, the system is always “telling” an entity “Believe S ”. How does the system tell an entity such a thing? If the entity is external to the system, it tells it by a certain pattern of outputs which the entity interprets as a belief. If the entity is internal, it may be told by internal communication events. If the

entity being “told” is not a separate process at all, but it is just part of the state of the system, its beliefs will be some function of that part of the state. For example, a file might be modeled simply as a component of the state of the entire system. The data in the file may be interpreted as beliefs, and the system writing into the file may be interpreted as the system “telling” the file “entity” something. This way of telling an entity something would be reflected simply as a change in system state which affects the component which represents the entity.

Whether an entity is told something by outputs, internal communication events, or state changes, there is a function which takes a finite trace of the system and returns a belief about the system for a given entity. (One might think that it would be more general to return a *set* of beliefs about the system, but a set of beliefs can always be replaced by the conjunction of those beliefs into a single belief). Telling the entity something just means making a transition which changes the value of this function, that is, which changes the entity’s beliefs.

Finally, what does it mean for an entity’s beliefs to be true? Suppose B is the function which takes a finite trace of the system and returns the belief that that entity has at the end of the trace. Given a finite trace of the system t , $B(t)$ is true if and only if $t \in B(t)$.

We are now ready to formally define what we mean by a true belief integrity policy for a system, and define when the system satisfies such an integrity policy (the definition of integrity).

Definition 1: Given a state machine, a *belief about* the system is a set of finite traces of the system. A *belief function* for the system is a function which takes a finite trace and returns a belief about the system. A *true belief integrity policy* (or *true belief policy* for short) for the system is a set of belief functions.

□

Definition 2: Given a state machine and a true belief policy P for the state machine, the system is said to *satisfy* the policy if and only if for every $B \in P$ and every finite trace t of the system, $t \in B(t)$.

□

We will often state true belief policies in the following way:

- We define a language L of statements in which an entity's beliefs may be expressed.
- We describe a semantics for L which specifies, for a given statement S in L , whether S is true for a given finite trace of the system.
- We describe the belief function for an entity by giving a definition of the set of statements of L that the entity believes by recursion on finite traces. In other words, we specify (1) what set of statements the entity believes initially, and (2) how the set of statements the entity believes changes in response to each possible state transition of the system.

If we can define the true belief policy using the above approach, it may be possible to prove the true belief policy for the system by induction on finite traces. In other words, we would show that (1) each entity's initial beliefs are true, and (2) if each entity's beliefs are true before a state transition, they are all true afterwards.

We will see examples of true belief policies in section 3.1.5. We will also see an example of a policy which is stated using the notions of true belief policies, but which does not have the form of a true belief policy.

3.1.4.4 Probabilistic Inference

In this section we describe a probabilistic version of the deducibility security model, and give an example to show how it can be applied. Before we give the definition of probabilistic deducibility security, we should mention why we are looking at probabilistic inference and security under the heading of theories of integrity. The reason is that many methods for ensuring integrity, such as passwords for authentication integrity and cryptographic methods, rely on certain keys remaining secret. It is generally not possible to protect these keys absolutely, but the methods for discovering them all use probabilistic inference. For example, codebreaking relies on regularities in the plaintext that are only probabilistic, and the breaking of these codes relies on guesses made on the basis of probabilistic reasoning using these regularities. In order to ensure certain kinds of integrity, therefore, it is necessary that we be able to analyze certain kinds of security, and that our analysis take probabilistic inference into account.

In nonprobabilistic deducibility security (as described in [56]), a secure system is described by a collection of *possible worlds*, i.e. possible execution histories of the system, and a set of *information functions* which represent the views of the system seen by various users and the information which should not be deducible by various users. A user with view v tries to deduce things about an information function i which he is not allowed to see directly by considering the set of possible worlds which are consistent with the observed value of v and the possible values of i in those worlds. The system is said to be secure if nothing can be deduced about such an i from v .

To add probability to deducibility security, we require that a collection of probability measures be put on the set of execution sequences. In section 3.1.3.2, we described how we associate probabilities with a state machine. The method for generating a set of probability measures on the set of execution sequences is described in Appendix A. For each probability measure, the system is treated as a probabilistic game in which users with a view v can use a certain strategy to try to guess facts about an information function i that they are not supposed to have access to. Probabilistic deducibility security essentially says that a system is secure if, for any yes/no question about the value of i , the users with view v can't guess the answer to the question with any greater probability than if they were guessing "blind", that is, without seeing the system at all. We will now make this precise.

Denote the set of all execution sequences of the probabilistic state machine under consideration by E , and let μ be one of the probability measures on E that are generated as described in Appendix A. Fix information function i and view v . The function i will have domain E , and range some set I . The function v will have domain E and range some set V . A yes/no question about the value of v can be represented as a subset Q of I (the question corresponding to Q is "Is the value of v in the execution sequence we're in an element of Q ?"). We first want to say what we mean by the probability of guessing the answer to question Q "blind". The users of the system are assumed to know what the possible execution sequences of the system and the probability measure μ are. Guessing "blind" means trying to guess, for an arbitrary execution sequence e , whether $i(e)$ is in Q or not. A probabilistic strategy for making such a blind guess has the form "guess that $i(e) \in Q$ with probability p " where p is a number between 0 and 1. Given such a strategy, what is the probability that it guesses correctly? There are two cases in

which this strategy guesses correctly:

1. $i(e) \in Q$ and the strategy guesses "yes": the probability of this case is the product of (1) the probability that $i(e) \in Q$, and the (2) the probability that the strategy guesses "yes". The first probability is

$$\mu(\{e \in E \mid i(e) \in Q\})$$

Call this number q_{yes} . The second probability is p . The probability of this case is therefore pq_{yes} .

2. $i(e) \notin Q$ and the strategy guesses "no": the probability of this case is the product of (1) the probability that $i(e) \notin Q$, and the (2) the probability that the strategy guesses "no". The first probability is $1 - q_{\text{yes}}$. The second probability is $1 - p$. The probability of this case is therefore $(1 - p)(1 - q_{\text{yes}})$.

The probability of guessing correctly in one of the two cases is the sum of the probabilities of being in each case separately, i.e.

$$(pq_{\text{yes}}) + (1 - p)(1 - q_{\text{yes}})$$

which can be rearranged to yield

$$(2q_{\text{yes}} - 1)p + (1 - q_{\text{yes}})$$

By our assumptions, the low users know the value of q_{yes} . We further assume that, whatever the value of q_{yes} , the low users will choose their strategy (in other words, they will choose the value of p) in such a way as to maximize their chance of guessing right. What value of p will maximize the expression above?

If $q_{\text{yes}} > .5$, the coefficient of p will be positive, so the probability of guessing blind correctly will be maximized by letting p be as large as possible, i.e. 1. With this value, the probability of guessing correctly is q_{yes} .

If $q_{\text{yes}} < .5$, the coefficient of p will be negative, so the probability of guessing blind correctly will be maximized by letting p be as small as possible, i.e. 0. With this value, the probability of guessing correctly is $1 - q_{\text{yes}}$.

If $q_{\text{yes}} = .5$, the coefficient of p is 0, and the probability of guessing blind correctly will be .5 regardless of the value of p .

In each of these cases, the maximum probability for guessing blind correctly is the larger of the two numbers q_{yes} and $1 - q_{\text{yes}}$. We will denote this number by G_b . This, then, is the maximum probability of blindly guessing the answer to question Q correctly.

We now want to compute the probability of guessing the answer to question Q correctly knowing the value of v . A probabilistic strategy for making such a guess has the form “seeing the value $v(e)$ guess that $i(e) \in Q$ with probability $s(v(e))$ ” where s is a function which takes an element of V and returns a number between 0 and 1. Given such a strategy, what is the probability that it guesses correctly? There are two cases in which this strategy guesses correctly:

1. $i(e) \in Q$ and the strategy guesses “yes”. The probability of being in this case is gotten by integrating the function s over the set of execution sequences e in which $i(e) \in Q$. The set being integrated over is

$$\{e \in E \mid i(e) \in Q\}$$

We will denote it by Q_{yes} . The probability of this case is therefore

$$\int_{e \in Q_{\text{yes}}} s(v(e)) d\mu(e)$$

2. $i(e) \notin Q$ and the strategy guesses “no”. The probability of being in this case is gotten by integrating the function $1 - s$ over the set of execution sequences e in which $i(e) \notin Q$. The set being integrated over is $E - Q_{\text{yes}}$. The probability of this case is therefore

$$\int_{e \in E - Q_{\text{yes}}} (1 - s(v(e))) d\mu(e)$$

The probability of guessing correctly in one of the two cases is the sum of the probabilities of being in each case separately, i.e. the sum of the two integrals above. Call this sum $G_{\mu,s}$. We say that a probabilistic state machine is *probabilistically deducibility secure* if and only if for every pair v and i where v is a view of users at level l and i is the information hidden from users at that level, and for every probability measure μ on the set of execution sequences, and for every set Q of values for i , and for every strategy s for guessing whether the value of i is in Q , the value $G_{\mu,s}$ is $\leq G_b$. Intuitively, what this says is that the low users at a given level cannot probabilistically guess the answer to any yes/no question about the high inputs any better using the low view than they can guessing blind. This game-theoretic definition of security is consistent with classical information theory, which examines how well a user at one end of a noisy channel can “guess” what was input by a user at the other end of the channel.

We will finish this section with a simple example of applying probabilistic inference to analyzing encryption. As with our other examples, we will not give all the formal details.

The system we wish to analyze is a simple channel over which data is being passed in encrypted form. The data is sent in units called *messages*. The set of messages is a finite set M with m elements. The encryption algorithm is simply a randomly chosen permutation e of the set M (since there are only finitely many such permutations, it is well-defined to say the permutation is chosen at random). The hidden information is the sequence of plaintext messages. The low view is the sequence of encryptions of the plaintext. (It is simple to make this system into a probabilistic state machine, but it would add nothing to the exposition to do so, so we will just talk about the information functions and not worry about how the system which generates them is modeled).

First, suppose that the probability distribution on the occurrences of plaintext messages is uniform, that is, every element of M has a $1/m$ chance of coming up each time a new plaintext message is sent out. Suppose the question we are trying to guess the answer to is a question about the n^{th} plaintext message, that is, there is some set C of plaintext messages such that the value of the high information is in the question set Q if and only if the n^{th} message is in C . We can easily show that if the number of elements

of C is c , then

$$G_b = \max((c/m), (1 - (c/m)))$$

Fix a strategy s for guessing the answer to question Q . Suppose s_{yes} is the probability of the strategy s guessing “yes”. Since the probability distribution on the n^{th} ciphertext can easily be shown to be independent of the probability distribution on the n^{th} plaintext, the integrals in $G_{\mu,s}$ can easily be evaluated to obtain

$$G_{\mu,s} = s_{\text{yes}}(c/m) + (1 - s_{\text{yes}})(1 - (c/m))$$

This expression is maximized by choosing s so that $s_{\text{yes}} = 1$ if $c > m/2$, and by choosing s so that $s_{\text{yes}} = 0$ if $c \leq m/2$. In either case, the value of $G_{\mu,s}$ turns out to be G_b , so no strategy can do better than guessing blind. Thus, if the plaintexts are randomly distributed and the key is randomly chosen, nothing can be deduced about any given plaintext.

Now, however, consider a question which involves more than one plaintext. For example, suppose the question is “are the first two plaintexts the same?”. We can choose a strategy which looks at the low view and guesses “yes” with probability 1 if the first two ciphertexts are the same, and guesses “no” with probability 1 if the first two ciphertexts are different. It is easy to show that this strategy always guesses the correct answer to the question, i.e. $G_{\mu,s} = 1$, and that $G_b = 1 - 1/m^2$, so it is possible to guess the answer to this question better from seeing the low view than guessing blind. This essentially demonstrates that while encryption may hide the contents of any given message perfectly, simple encryption will not encrypt *extended* patterns of plaintexts.

3.1.5 Examples

In this section we describe one of the examples we’ve looked at to drive the formulation of our theories of integrity. This example is described semifor-
mally. Other examples can be found in Volume III, the library of models.

3.1.5.1 Distributed Database

In this section we describe a distributed airplane database example. The database maintains a schedule of airplane flights. This schedule is stored in a central location, and is queried and updated from multiple terminal interfaces. Users make requests to schedule or cancel flights, and receive reports on the system's response. Users interpret the reports as information about database state. The integrity requirement is that users' beliefs, based on system reports, are true.

We considered four versions of the database. Each version has the same integrity policy, but the different versions use slightly different measures for ensuring integrity. The reason for having four versions was to test whether the integrity policy correctly captured our informal notions of integrity for a distributed database, and whether the policy "recognized" certain measures for ensuring integrity. To this end, we deliberately designed two of the four versions with "integrity flaws". The other two versions used standard methods to "fix" the flaws. Our analysis showed that the two "flawed" versions failed to satisfy the integrity policy, whereas the two "fixed" versions did. These results validate the use of true belief policies to specify certain kinds of integrity.

We will first describe the elements of the database which are common to all four versions. First, there is a fixed set of planes and destinations. For simplicity, there are only two kinds of input: commands to schedule a flight, and commands to cancel a flight. A scheduling command takes place at a certain location (i.e. terminal), and specifies a plane, a departure time, an arrival time, and an arrival location. There is only one kind of output, namely action reports. Action reports take place at a certain location, and specify whether a particular command was acted on or not. (Action reports could also contain the information of why a command was not acted on if it was not, but this would not affect our analysis).

There is a function *sched* which takes a state of the system and returns the schedule for the planes. The schedule just consists of times and places of arrival and departure for each plane. The value of *sched* must satisfy certain constraints, including the following:²

²These constraints themselves are a form of integrity policy, in that they are require-

- The time between the departure of a plane for a destination and the arrival time for that destination must be long enough for the plane to make the trip, and short enough to allow it to do so without refueling (refueling stops count as arrivals).
- The departure of a plane must be from the last place it arrived.
- There cannot be two departures in a row or two arrivals in a row for a given plane.
- The time between arrival and departure for a plane must be long enough that the plane can be serviced.

Informally, the operation of the database is to accept scheduling commands from terminals and make the update directed by the command unless (1) the command doesn't make sense, for example, it is a cancellation of a flight that is not scheduled; (2) making the update would put the system in a state in which the value of *sched* would fail to satisfy one of the above constraints; or (3) the command is a cancellation of a flight which was scheduled from another terminal (this last requirement is just to simplify the belief policy we will state in a moment). Whether the command is acted on or not, the database sends a report through the terminal from which the command came saying whether it was acted on or not.

For each terminal, we can give a recursive definition of a function which takes a finite trace and returns the set of commands from that terminal to schedule flights which have been acted on, and which we have not sent a cancel command for. When an output is received that a command to schedule a flight has been acted on, that command is added to the set. When a command is issued to cancel a command in the set, that command is removed from the set. It remains out of the set unless an output comes back saying the cancellation has not been acted on, in which case it is put back in the set. The set of commands currently in the set can be interpreted as a set of statements about the current schedule in the obvious way, and so can be interpreted as a set of statements about the current trace (namely, that the value of *sched* on the state at the end of the current trace contains

ments that the database be consistent with the real world, but they are not the sort of integrity requirements we are interested in here.

all the flights corresponding to commands in the set). This gives rise to a belief function for each terminal. This set of belief functions constitutes the true belief policy for the database.

The differences between the versions of the database are in how they handle multiple overlapping transactions. We will now describe the different versions.

In version 1, each database update spawns an update process which (1) checks to see if the update would make the database inconsistent, (2) updates database if it wouldn't, and (3) reports back. Each of the 3 steps above are assumed to be atomic, but the sequence of them is not. A state machine for this version would have a component of state for each currently active update process. This state component would include the information of what update the process was doing, whether it had performed the consistency check, what the result was if it had, and whether it had done the update.

Even without writing down a formal specification of this version of the database, we can give a semiformal argument that it won't satisfy the integrity policy because transactions are not serializable. Consider the following scenario:

1. A user at terminal A issues a command to schedule plane P to fly to destination D at time T. Suppose the state of the database is such that it would be consistent to act on this update.
2. A user at terminal B ($\neq A$) issues a command to schedule plane P to fly to destination E ($\neq D$) at time T. Suppose it would also be consistent to act on this update.
3. The update process for the first command is spawned, checks the update for consistency, and finds it consistent.
4. The update process for the second command does the same.
5. The update for the first command is made, and the result reported to terminal 1.
6. The update for the second command is made, overwriting the previous update, and the result is reported back to terminal 2.

At the end of this sequence, the set of traces that terminal 1 users believe they are in does not include the actual current trace, so the integrity policy is violated.

In version 2 of the database, update processes lock the database variables until update and report are done. A state machine for this version would have the additional information of a boolean value for each part of the database that could be independently locked. For simplicity, we'll just imagine that update processes just lock the entire database, so there's only one additional boolean value in the state of version 2. The state component for an update process would include the additional information of whether the update process had locked the database, and whether it had unlocked it. Update processes start up in the state of not yet having locked the database, and can only proceed past that state if the database is unlocked.

This system satisfies the integrity policy because locking ensures serializability. The proof of this would require that we prove certain state invariant by induction on traces. This invariant would include the following clauses:

1. The current state of the database, plus any updates which update processes have checked for consistency but not yet performed, is consistent.
2. There is at most one update process which is not waiting to lock the database, and if there is one, then the database is locked.
3. Any reports which an update process is about to make are consistent with the current state of the database.

This invariant is satisfied because of the way locking functions. The last clause of the invariant allows us to show that the belief based on commands and reports at a given terminal are always true, so the integrity policy is satisfied. If we tried to prove a similar invariant for version 1, the proof would fail because clause 1 can be true in one state, but false after a state transition in which an update process checks its update for consistency with the database and finds it consistent with the database, but not with the other pending updates. In version 2, if a process is checking an update for consistency, it must be the sole process doing so (by clause 2 of the invariant), so checking against the database is sufficient to maintain consistency.

In version 3 of the database, the terminals through which updates are entered and reports received are *remote* in the sense that an update process *sending* a report and the terminal *receiving* that report are separate events. In this version, processes send off their action report, unlock the database, and exit, but the report does not reach the terminal until later. A state machine for this version would be like that for version 3, except that its state would include the additional information of what action reports were on their way to which terminals.

Again, even without writing down a formal specification of this version of the database, we can give a semiformal argument that it won't satisfy the integrity policy because reports can arrive in a different order than the commands they report on were processed. Consider a situation in which a terminal sends a command to schedule a flight, then one to cancel that flight, then a third command to schedule it after all. Suppose the reports for these commands come back in the same order in which the commands were sent, each command reporting success. The value of the belief function for such a trace includes the belief that the flight in question is now scheduled. However, if the two commands to schedule the flight were processed first, then the command to cancel, and the report on the cancellation passes the report on the second schedule command on the way back to the terminal, the flight will actually be cancelled. In other words, the belief function for that terminal will not contain the current trace, in violation of the integrity policy.

In version 4 of the database, the database and the remote terminals run a simple agreement protocol. The remote terminals acknowledge receipt of reports. Update processes keep the database locked until the acknowledgement is received. A state machine for this version would have the additional information of where the terminals and the update processes were in the protocol, and what messages and acknowledgements had been sent but not yet received.

This system satisfies the belief policy because the acknowledgement mechanism ensures that reports are always current. The proof of this would be similar to that for version 2, with the invariant having the additional clause that any reports or acknowledgements in transit are associated with an update process which is not waiting to lock the database. With the other clauses

of the invariant of version 2, this ensures that reports arrive in the same order that their corresponding commands are processed, so the integrity policy is satisfied.

3.2 Authentication Protocols

In this section we are not concerned with general integrity. We concentrate here on a particular part of integrity that is vital to certain situations. Authentication protocols are presented in the National Computer Security Center publication, “Integrity in Automated Information Systems” [33], as a mechanism for establishing identity and for supporting encryption. Among the aims of these protocols is the distribution of encryption keys, which is needed for cryptographic protection of data, another part of integrity. A typical aim of a protocol is to establish an encryption key shared by two principals. The message exchange often involves a third party — a “keyserver” — who is trusted to generate good keys and keep secrets. Protocol messages employ a variety of techniques to ensure the identity of a principal, that the messages have been recently generated, and that the keys exchanged are protected.

In recent years, several logics of authentication have been developed that use *belief* logics [18, 9]. These logics enable the user to reason about the beliefs of the “principals” (i.e., the various processes) involved in a protocol. Typical beliefs are that “this key is good”, that “this message is fresh”, that “a particular principal really sent this message”, or that “a particular principal is trustworthy”. To build the logic, inference rules from the problem domain are formalized, as is the relation between protocol messages and beliefs. This section presents an authentication logic that is based on [18] and enriched by several new constructs. A tool for the verification of authentication protocols based on the implementation of this logic is described in Volume IV, the Romulus User’s Manual.

3.2.1 Authentication

There is a conventional way in which cryptographic protocols are documented in the literature. A protocol is often described as a sequence of messages. Each message consists of a message header, which describes the source and destination of the message, and a message body, which represents the data to be transmitted. For convenience of reference, each message is normally preceded by a sequence number.

Such a protocol description clearly targets an audience of implementors, but is quite inadequate in expressing what a protocol is or is not supposed to do (for example, why the protocol is secure). Thus one always finds a corresponding text explanation (for example, in the English language) of the working of a protocol. To analyze a protocol, we need to rigorously specify both the messages and their meanings in an unambiguous way.³

Therefore, we would like to document a protocol in two ways, one for implementors and one for analyzers. For analyzers, we incorporate implementors' descriptions into a logic. This logic is implemented within the logic of the HOL theorem prover. In section 3.2.2 we describe the conventions used to describe protocols for implementors; In section 3.2.3 we discuss the logical statements used to describe protocols for analysts. In section 3.2.4 we describe the HOL implementation of the whole logic.

3.2.2 Protocol Descriptions

This section describes the conventions that are commonly used (with minor variations) to describe cryptographic protocols in the literature. We present this to give the reader background in conventional notation, which we use from time to time in our discussion. We implement this language, translated, in HOL, which is described in section 3.2.4.

A protocol consists of a series of messages in a particular form. A message consists of a header and a body. The header tells us who is sending the message and who is receiving the message. The source (or destination) of a message is denoted by the identifier of its sender (or of its intended recipient).

³There is at least one published protocol where the authors wrongly explained that the protocol was secure.

Communicating parties such as the sender or the recipient are also called *principals*. A principal's identifier is denoted by a string beginning with a capital letter. We use the arrow \rightarrow to denote that a message is being sent from one principal to the other. Thus, the header of a message sent from principal A to principal B is simply

A \rightarrow B

where the space is a separator. The data item that is sent in a message is called a *message body* and is sometimes simply referred to as a message when there is no ambiguity. This data item is denoted by a mathematical expression describing how it is computed. Any identifier (i.e., a string beginning with a letter, thus including a principal identifier) can be a data item by itself. As an example, suppose x and y are data items, then their bit-wise exclusive-or is also a data item, which can be denoted by

$x \text{ XOR } y$.

A list (similar to concatenation) of two data items forms a new data item by joining with the operator $,$:

x, y .

Encryption of a data item with another data item as an encryption key also results in a data item:

$\{x\}_e(k)$

where x is the plaintext, k is the key, and $\{x\}_e(k)$ is the ciphertext. Note that $($ and $)$ also serve as separators. Decryption is denoted in a similar fashion:

$\{x\}_d(k)$.

These encryptions and decryptions represent conventional algorithms such as the Data Encryption Standard (DES). To represent public key algorithms such as the RSA system, we use

$\{x\}_{pe(k)}$

and

$\{x\}_{pd(k)}$.

According to the properties of encryption, expression $\{\{x\}_e(k)\}_d(k)$ is equivalent to x . If k_1 and k_2 are a pair of corresponding public and private keys, then expression $\{\{x\}_{pe(k_1)}\}_{pd(k_2)}$ is equivalent to x . For public key systems like RSA, expression $\{\{x\}_{pd(k_2)}\}_{pe(k_1)}$ is equivalent to x .

The header and the body of a message are separated by a ':', thus A sending $\{x\}_e(k)$ to B is described as

A → B: $\{x\}_e(k)$

A protocol is a sequence of messages to be exchanged in the order as they are listed. The message are separated by the operator ';'.

The familiar Denning-Sacco key distribution protocol can thus be described as:

1. A → S: A, B;
2. S → A: {B, Kab, Ts, {A, Kab, Ts}_e(Kbs)}_e(Kas);
3. A → B: {A, Kab, Ts}_e(Kbs);

Here messages are preceded with sequence numbers for reference purposes. In this protocol, three messages are sent among principals A, B, and S. S is a trusted key server and A and B are clients who wish to obtain a secret key they can use for encryption of messages between them. For clients A and B, Kab is the session key chosen by the server S; Ts is the timestamp taken from the server's clock. A and S share the secret key Kas, and Kbs is the secret key shared between B and S.

3.2.3 Protocol Specifications

As we said before, a protocol description is sufficient for a protocol implementor, but not adequate for the purpose of analyzing protocols. We will use

a logic of authentication to analyze the protocol and see if it could achieve its design objectives. This logic is essentially the logic by Burrows, Abadi, and Needham, but enhanced with a few constructs suggested by Gong, Needham, and Yahalom. This logic is the subject of this section.

Initial assumptions, final goals, and message meanings are all some form of logical statement in the logic of authentication. We first describe what kinds of logical statements the user can make, and then we describe how to put them in a protocol specification.

Several other entities, needed for the constructing and sending of messages, along with the inference rules, are introduced in the following section, which presents the HOL version of the whole system. The key notion of statements warrants a separate introduction here.

Often the fact that a principal sends a data item x reflects the principal's current state of mind. A statement (say s) about this state could be attached to the data item as an extension to denote the *meaning* of sending the data item. We need some way to attach extension statements to the message text they apply to. We use a function **extension**, which maps message text to the extension statement, and define the extensions of message parts that way. It corresponds in a natural way to what extensions are—kinds of preconditions for the message to be sent, rather than objects actually sent with a message. A drawback, however, with this approach is that the one bit of message text cannot mean different things in different positions in the messages. This is not a problem in any of the protocols encountered. A full solution is to include information in the extension function about the particular occurrence intended, and we will do this in a future version.

Suppose p and q are principal identifiers and x is a data item, then the following are all logical statements that we explain one by one.

send p q x . Principal p *sends* data item x to principal q .

receive p x . Principal p *receives* data item x , possibly after performing some computation such as decryption. That is, a data item received can be the data item itself or an item that is feasible to compute by p using its own resources together with x .

possesses p x . Principal p *possesses* data item x . Principal p is able to

repeat this data item in future messages in the current session. At any particular stage of a session, p possesses the data items p initially possessed when the session began and the data items p has received so far. The data items a principal generates during a session are considered to be included in the principal's initial possessions. In addition, p possesses a data item that is feasibly computable from the data items p already possesses.

convey p x . Principal p *conveyed* data item x . Such an item can be a data item explicitly exchanged or some feasibly computable (by the intended recipient of the data item) content of such a data item. Thus a data item can also be conveyed implicitly.

is_fresh x . Data item x is *fresh*. It has never been used in a previous message. An example is a "nonce" — a random number generated for the purpose of being fresh.

is_recog x . Data item x is *recognizable*. An intended recipient would recognize the data item if the recipient has certain expectations about its contents before actually receiving it. The recipient may recognize a particular value (for example, the recipient's own identifier), a particular structure (for example, the format of a timestamp), or other forms of redundancy.

is_shared_secret p q x . Data item x is a suitable *secret* for principals p and q . They may properly use x to prove each other's identity. They may also use it as (or derive from it) an *encryption key* for secure communication. This notation is symmetrical in that **is_shared_secret p q x** and **is_shared_secret q p x** are equivalent.

By default, we assume that secrets will never be discovered by any principal except the legitimate owners or principals the owners trust. In the latter case, the trusted principals never use the secret as a proof of identity or as an encryption key.

Suppose s is a statement, then the following are also statements:

believes p s . Principal p *believes* that statement s holds. If s is an empty statement, then so is **believes p s** .

`elig p x`. Principal `p` is *eligible* to convey data item `x`. `p` possesses the data item and believes that the statement expressed as the extension of the data item holds.

`believes p (juri q s)`. Principal `p` believes that principal `q` has *jurisdiction* over statement `s`. Principal `p` believes that `q` is an authority on `s` and should be trusted in this respect.

`believes p (juris_star q)`. Principal `p` believes that principal `q` has jurisdiction over all `q`'s beliefs. That is, principal `q` is considered by `p` to be honest and competent.

Also, statements can be joined by standard logical operators such as `/\` (meaning AND) and `\|` (meaning OR) to form new statement.

3.2.4 The crypto_90 Theory

In this section we give an annotated description of the `crypto_90` theory as implemented in the file `crypto_90.sml`. The notions of the previous sections are translated in a fairly straightforward way into the HOL setting. Also, in this section, we present the inference rules of the logic. The `crypto_90.sml` file starts with some standard commands to remove old versions of the HOL theory.

```
System.Unsafe.SysIO.unlink "crypto_90.holsig"
  handle e => print "no earlier crypto_90.holsig to remove\n";
System.Unsafe.SysIO.unlink "crypto_90.thms"
  handle e => print "no earlier crypto_90.thms to remove\n";
```

To set up the proof environment, a number of new things need to be defined and/or declared in HOL. First, we need to enter draft mode (by starting a new theory) to do this.

```
new_theory "crypto_90";
```

The function `APP` is defined here as an infix variant of the HOL `APPEND` function.

```

new_infix_definition(
  "APP_def",
  --'APP (l1:('a)list) (l2:('a)list) = APPEND l1 l2'--,
  600);

```

Now we will build theory `crypto_90` once, and later the user would simply load it in. To implement the logic within HOL, we have two main goals. One is to define the statements of the logic: these are the objects of type `statement`. The other is to define the turnstile of the logic: the predicate `theorem`. Statements mapped to the HOL Boolean value `true` by `theorem` are the theorems of our logic.

The following declares a type for principals, another for text (plaintext or ciphertext), and one more for (logical) statements.

```

new_type{Name= "principal", Arity= 0};
new_type{Name= "text", Arity= 0};
new_type{Name= "statement", Arity= 0};
val textlist = ty_antiqu(':(text)list'==);

```

The current release of HOL90 unfortunately does not support type abbreviations; as a work-around, we define the SML variable `textlist` as above and use `^textlist` in the logic to refer to the intended type.

The function `name` is used to designate the name of a principal in a form that can be included in messages.

```

new_constant{Name = "name",
  Ty = ==':principal -> ^textlist'==};

```

Encryption takes the first argument as the key and encrypts the second argument to yield a piece of text. We define functions `encrypt` and `decrypt` for encryption and decryption. Likewise, we define `pencrypt` and `pdecrypt` for public key encryption and decryption algorithms.

```

new_constant{Name = "encrypt",
  Ty = ==':^textlist -> (^textlist -> ^textlist)'==};
new_constant{Name = "decrypt",
  Ty = ==':^textlist -> (^textlist -> ^textlist)'==};
new_constant{Name = "pencrypt",

```

```

      Ty = ==': ^textlist -> (^textlist -> ^textlist)'==};
new_constant{Name = "pdecrypt",
      Ty = ==': ^textlist -> (^textlist -> ^textlist)'==};

```

The function `feas` represents a known one-to-one function such that the function and its inverse are feasible to compute.

```

new_constant{Name = "feas",
      Ty = ==': ^textlist -> ^textlist'==};

```

In the HOL version of the logic all messages must have an extension. Some inference rules require belief in the extension of a message, in the hypothesis, and it is convenient to define the distinguished statement `nil`. This statement corresponds loosely to “true”.

```

new_constant{Name = "nil",
      Ty = ==': statement'==};

```

Certain properties may hold for a piece of text. For example, it can be “fresh” or “recognizable”. It can be a “shared secret” between two principals. A principal can have “conveyed” a text, or might “possess” it. These and other properties mentioned in the previous section are declared, and later, axioms are stated about their properties. In particular, we define the operator of “belief”.

```

new_constant{Name = "possesses",
      Ty = ==': principal -> (^textlist -> statement)'==};
new_constant{Name = "convey",
      Ty = ==': principal -> (^textlist -> statement)'==};
new_constant{Name = "elig",
      Ty = ==': principal -> (^textlist -> statement)'==};
new_constant{Name = "is_fresh",
      Ty = ==': ^textlist -> statement'==};
new_constant{Name = "is_recog",
      Ty = ==': ^textlist -> statement'==};
new_constant{Name = "is_shared_secret",
      Ty = ==': principal ->
          (principal -> (^textlist -> statement))'==};
new_constant{Name = "believes",
      Ty = ==': principal -> (statement -> statement)'==};

```

```

new_constant{Name = "juris",
              Ty == ':principal -> (statement -> statement)'==};
new_constant{Name = "juris_star",
              Ty == ':principal -> statement'==};
new_constant{Name = "receive",
              Ty == ':principal -> (~textlist -> statement)'==};

```

For extensions, we have:

```

new_constant{Name = "extension",
              Ty == ':~textlist -> statement'==};

```

After these declarations and more ground work, we need to state all the inference the rules of the logic. We declare the turnstile of our logic, as mentioned at the beginning of this section.

```

new_constant{Name = "theorem",
              Ty == ':statement -> bool'==};

```

We declare `send` for sending messages from one principal to another.

```

new_constant{Name = "send",
              Ty == ':principal ->
                    (principal -> (~textlist -> bool))'==};

```

Next we define laws regarding symmetric and public key encryption.

```

new_open_axiom("Y1", -- '!x k. decrypt k (encrypt k x) = x'--);
new_open_axiom("Y2", -- '!x k. pdecrypt k (pencrypt k x) = x'--);
new_open_axiom("Y3", -- '!x k. pencrypt k (pdecrypt k x) = x'--);

```

The first set of inference rules is about receiving messages

```

new_open_axiom("R1", -- '!p q x.
                    send p q x /\
                    theorem (elig p x)
                    ==> theorem (receive q x)'--);

```

Rule R1 says that if a protocol specifies a message in which p sends x to q , and p is eligible to convey such a data item, then q receives x . This rule prevents the specification of protocols that are infeasible to implement. It is a feature of the logic we use here and distinguishes our logic from previously published logics. The eligibility concept was first discussed by Gong, in [16]. This rule relates the implementable protocol messages to the logic and thus is the basis of the formal semantics of protocols.

```
new_open_axiom("R2", --'!p x y.
               theorem(receive p(x APP y))
               ==> theorem(receive p x)'--);
new_open_axiom("R3", --'!p x y.
               theorem(receive p(x APP y))
               ==> theorem(receive p y)'--);
```

Rules R2 and R3 say that receiving a list implies receiving each member in the list. (There is no rule R4 owing to a typographical oversight that will be corrected in the next release.)

```
new_open_axiom("R5", --'!p x k.
               theorem(receive p (encrypt k x)) /\
               theorem(possesses p k)
               ==> theorem(receive p x)'--);
```

Rule R5 says that if p receives an encrypted item and if p also possesses the encryption key then p has received the corresponding plaintext.

```
new_open_axiom("R6", --'!p x.
               theorem(receive p (feas x))
               ==> theorem(receive p x)'--);
```

Rule R6 says that if p receives some feasibly computable invertible function of x , then p has received x .

The next set of inference rules is about possession.

```
new_open_axiom("P1", --'!p x.
               theorem(receive p x)
               ==> theorem(possesses p x)'--);
```

Rule P1 says that a principal possesses a received item.

```
new_open_axiom("P2", --'!p x y.  
    theorem(possesses p x) /\  
    theorem(possesses p y)  
    ==> theorem(possesses p(x APP y))'--);
```

Rule P2 says that a principal who possesses and item x and possesses an item y, also possesses the list of x and y.

```
new_open_axiom("P3", --'!p x y.  
    theorem(possesses p(x APP y))  
    ==> theorem(possesses p x)'--);  
new_open_axiom("P4", --'!p x y.  
    theorem(possesses p(x APP y))  
    ==> theorem(possesses p y)'--);
```

Rules P3 and P4 say that a principal who possesses the list of x and y, also possesses x and y individually.

```
new_open_axiom("P5", --'!p x k.  
    theorem(possesses p x) /\  
    theorem(possesses p k)  
    ==> theorem(possesses p (encrypt k x))'--);  
new_open_axiom("P6", --'!p x k.  
    theorem(possesses p (encrypt k x)) /\  
    theorem(possesses p k)  
    ==> theorem(possesses p x)'--);  
new_open_axiom("P7", --'!p x k.  
    theorem(possesses p x) /\  
    theorem(possesses p k)  
    ==> theorem(possesses p (decrypt k x))'--);
```

Rules P5, P6, and P7 say that anyone can perform encryption and decryption, provided they possess the proper keys.

```
new_open_axiom("P8", --'!p x.  
    theorem(possesses p (feas x))  
    ==> theorem(possesses p x)'--);  
new_open_axiom("P9", --'!p x.  
    theorem(possesses p x)  
    ==> theorem(possesses p (feas x))'--);
```

Rules P8 and P9 says that if p possesses some feasibly computable invertible function of x , then p possesses x and vice versa.

The next set of rules are about freshness. A message is regarded as fresh if it is believed that a principal could not possibly generate the message before the current protocol execution.

```
new_open_axiom("F1", --'!p x y.
    theorem(believes p (is_fresh x))
    ==>
    theorem(believes p (is_fresh (x APP y)))'--);

new_open_axiom("F2", --'!p x y.
    theorem(believes p (is_fresh x))
    ==>
    theorem(believes p (is_fresh (y APP x)))'--);
```

Rules F1 and F2 say that if a principal p believes that item x is fresh, then p will believe that any list containing x will also be fresh.

```
new_open_axiom("F3", --'!p x k.
    theorem(believes p (is_fresh x)) /\
    theorem(possesses p k)
    ==>
    theorem(believes p (is_fresh (encrypt k x)))'--);
new_open_axiom("F4", --'!p x k.
    theorem(believes p (is_fresh x)) /\
    theorem(possesses p k)
    ==>
    theorem(believes p (is_fresh (decrypt k x)))'--);
```

Rules F3 and F4 say that encryption or decryption of fresh data will also be fresh.

```
new_open_axiom("F5", --'!p x k.
    theorem(believes p (is_fresh k)) /\
    theorem(possesses p k) /\
    theorem(believes p (is_recog x))
    ==>
    theorem(believes p (is_fresh (encrypt k x)))'--);
new_open_axiom("F6", --'!p x k.
    theorem(believes p (is_fresh k)) /\
```



```

theorem(possesses p k) /\
theorem(believes p (is_recog x))
==>
theorem(believes p (is_fresh (decrypt k x)))'--;

```

Rules F5 and F6 say that if p believes a key to be fresh and p can recognize a data item then p will believe that the encryption or decryption of that item with that key will also be fresh.

```

new_open_axiom("F7", --'!p x.
theorem(believes p (is_fresh x))
==>
theorem(believes p (is_fresh (feas x)))'--;

```

Rule F7 says that if p believes that an item is fresh then p will believe that any feasibly computable function of the item is fresh.

The next set inference rules have to do with recognizability.

```

new_open_axiom("G1", --'!p x y.
theorem(believes p (is_recog x))
==>
theorem(believes p (is_recog (x APP y)))'--;
new_open_axiom("G2", --'!p x y. theorem(believes p (is_recog x))
==>
theorem(believes p (is_recog (y APP x)))'--;

```

Rules G1 and G2 say that if p believes it can recognize item x, then p believes it can recognize any list that contains x.

```

new_open_axiom("G3", --'!p x k.
theorem(believes p (is_recog x)) /\
theorem(possesses p k)
==>
theorem(believes p (is_recog (encrypt k x)))'--;
new_open_axiom("G4", --'!p x k.
theorem(believes p (is_recog x)) /\
theorem(possesses p k)
==>
theorem(believes p (is_recog (decrypt k x)))'--;

```

Rules G3 and G4 say that if p believes it can recognize item x and possesses an encryption key, then p believes it can recognize an encrypted or decrypted

form of item x . (There is no rule G5 owing to a typographical oversight that will be corrected in the next release.)

```
new_open_axiom("G6", --'!p x.
               theorem(believes p (is_recog x))
               ==>
               theorem(believes p (is_recog (feas x)))'--);
```

Rule G6 says that if p believes it can recognize item x then p believes it can recognize any feasibly computable function of x .

The next inference rule states that a principal who believes that k is a shared secret between p and another principal q will also believe that k is a shared secret between q and p .

```
new_open_axiom("S1", --'!p q k.
               theorem(believes p (is_shared_secret p q k))
               ==>
               theorem(believes p (is_shared_secret q p k))'--);
```

The next set of inference rules are rules of conveyance. Rules in this section govern how a principal advances its beliefs by analyzing the messages it receives.

```
new_open_axiom("M1", --'!p q x k.
               theorem(receive p (encrypt k x))           /\
               theorem(possesses p k)                     /\
               theorem(believes p (is_shared_secret p q k)) /\
               theorem(believes p (is_recog x))           /\
               theorem(believes p (is_fresh(x APP k)))
               ==>
               theorem(believes p (convey q x))'--);
new_open_axiom("M2", --'!p q x k.
               theorem(receive p (encrypt k x))           /\
               theorem(possesses p k)                     /\
               theorem(believes p (is_shared_secret p q k)) /\
               theorem(believes p (is_recog x))           /\
               theorem(believes p (is_fresh(x APP k)))
               ==>
               theorem(believes p (convey q (encrypt k x)))'--);
new_open_axiom("M3", --'!p q x k.
               theorem(receive p (encrypt k x))           /\
```

```

theorem(possesses p k) /\
theorem(believes p (is_shared_secret p q k)) /\
theorem(believes p (is_recog x)) /\
theorem(believes p (is_fresh(x APP k)))
==>
theorem(believes p (possesses q k))'--);

```

Rules M1, M2, and M3 essentially reflect the fact that if you receive a properly encrypted data item that is also both fresh and recognizable, then you should believe that the other party who has the proper secret must be the sender and that the sender possesses the data item and the keys.

```

new_open_axiom("M4", --'!p q x.
  theorem(believes p (convey q x)) /\
  theorem(believes p (is_fresh x))
==> theorem(believes p (possesses q x))'--);

```

Rule M4 says that if p believes that q conveyed an item and that the item is fresh then p believes that q possesses that item.

```

new_open_axiom("M5", --'!p q x y.
  theorem(believes p (convey q (x APP y)))
==> theorem(believes p (convey q x))'--);

```

```

new_open_axiom("M6", --'!p q x y.
  theorem(believes p (convey q (x APP y)))
==> theorem(believes p (convey q y))'--);

```

Rules M5 and M6 say that if p believes that q conveyed a list of items then p believes that q conveyed each item in the list.

The next set of rules deal with jurisdiction.

```

new_open_axiom("J1", --'!p q s.
  theorem(believes p (juris q s)) /\
  theorem(believes p (believes q s))
==> theorem(believes p s)'--);

```

Rule J1 says that if p believes that q is an authority on statement s and that q believes s, then p should also believe s.

```

new_open_axiom("J2", --'!p q x s.
    theorem(believes p (juris_star q)) /\
    theorem(believes p (convey q x)) /\
    theorem(believes p (is_fresh x)) /\
    (extension x = s)
    ==> theorem(believes p (believes q s))'--);

```

Rule J2 says that if a principal sends a fresh data item, then the sender should believe in the statement specified in the extension of the item, if the sender is honest and competent.

```

new_open_axiom("J3", --'!p q s.
    theorem(believes p (juris_star q)) /\
    theorem(believes p (believes q (believes q s)))
    ==> theorem(believes p (believes q s))'--);

```

Next come rules of eligibility. This category determines what items a principal is eligible to convey according to the items it possesses and the beliefs it holds.

```

new_open_axiom("E1", --'!p x.
    theorem(possesses p x)
    ==> theorem(elig p x)'--);

```

Rule E1 says that a principal is obviously eligible to repeat in future messages a data item that it possesses. Recall that x ranges over data items with or without extensions.

```

new_open_axiom("E2", --'!p x k.
    theorem(elig p x) /\
    theorem(possesses p k) /\
    theorem(believes p (is_shared_secret p q k)) /\
    theorem(believes p (extension x))
    ==> theorem(elig p (encrypt k x)) /\
    theorem(elig p (decrypt k x))'--);

```

Rule E2 says that a principal can convey its own belief by sending a properly encrypted message.

```

new_open_axiom("E3", --'!p x y.
               theorem(elig p x) /\
               theorem(elig p y)
               ==> theorem(elig p (x APP y))'--);

```

Rule E3 says that if p is eligible to send x and is eligible to send y then p is eligible to send the list of x and y .

Next, we have an axiom that says that everyone believes `nil`. This statement is often defined as the extension of a message object.

```

new_open_axiom("X1", --'!p. theorem(believes p nil)'--);

```

Axiom E2 gives sufficient conditions under which a principal is eligible to perform encryption and decryption. We often want to use just the part about eligibility to *encrypt*, and so we prove a theorem that extracts this half of the axiom.

```

save_thm ("elig_encr", GEN_ALL (DISCH_ALL (CONJUNCT1 (UNDISCH
(SPEC_ALL (axiom "crypto_90" "E2")))))));

```

We expect to add further such theorems to the theory `crypto_90` over time.

Finally, the theory is exported and HOL is exited.

```

export_theory();
exit();

```

3.2.5 Analyzing a Protocol

To analyze a protocol in the implemented theory, we first declare the various objects (for example, principals) involved in the protocol. Then the messages are defined in HOL form. We only mention here what is needed to show the structure of the theories and the proving environment. For example, the messages of the Denning-Sacco protocol are stipulated by the axioms:

```

new_open_axiom("dsm1", --'send A Svr ((name A) APP (name B))'--);
new_open_axiom("dsm2", --'send Svr A (encrypt Kas ((name B) APP Kab APP
Ts APP (encrypt Kbs ((name A) APP Kab APP Ts))))'--);
new_open_axiom("dsm3", --'send A B (encrypt Kbs ((name A) APP
Kab APP Ts))'--);

```

The initial assumptions are given as axioms, also.

```

new_open_axiom("dsa1", --'theorem(believes A
    (is_shared_secret A Svr Kas))'--);
new_open_axiom("dsa2", --'theorem(believes A (is_fresh Ts))'--);
new_open_axiom("dsa3", --'theorem(believes A (is_recog (name B)))'--);
new_open_axiom("dsa4", --'theorem(possesses A Kas)'--);
new_open_axiom("dsa5", --'theorem(possesses A (name A))'--);
new_open_axiom("dsa6", --'theorem(possesses A (name B))'--);

new_open_axiom("dsb1", --'theorem(believes B
    (is_shared_secret B Svr Kbs))'--);
new_open_axiom("dsb2", --'theorem(believes B (is_fresh Ts))'--);
new_open_axiom("dsb3", --'theorem(believes B (is_recog (name A)))'--);
new_open_axiom("dsb4", --'theorem(possesses B Kbs)'--);

new_open_axiom("dss1", --'theorem(possesses Svr Kas)'--);
new_open_axiom("dss2", --'theorem(possesses Svr Kbs)'--);
new_open_axiom("dss3", --'theorem(possesses Svr Kab)'--);
new_open_axiom("dss4", --'theorem(possesses Svr Ts)'--);
new_open_axiom("dss5", --'theorem(possesses Svr (name A))'--);
new_open_axiom("dss6", --'theorem(possesses Svr (name B))'--);
new_open_axiom("dss7", --'theorem(believes Svr
    (is_shared_secret Svr B Kbs))'--);
new_open_axiom("dss8", --'theorem(believes Svr
    (is_shared_secret Svr A Kas))'--);

```

The final conditions that we wish protocol execution to achieve are collected and named as the postcondition. For example:

```

new_definition ("postcond", --'postcondition =
    theorem(possesses A Kab) /\
    theorem(believes A (convey Svr ((name B) APP Kab APP Ts))) /\
    theorem(believes A (is_fresh ((name B) APP Kab APP Ts))) /\
    theorem(possesses B Kab) /\
    theorem(believes B (convey Svr ((name A) APP Kab APP Ts))) /\
    theorem(believes B (is_fresh ((name A) APP Kab APP Ts)))'--);

```

Proving the protocol is then a matter of proving that the postcondition follows from the axioms (the initial conditions and the message axioms). The inference rule that links the messages with the specification (the initial assumptions and the postcondition) is

```
new_open_axiom("R1", --'!p q x.  
                send p q x /\ theorem (elig p x)  
                ==> theorem (receive q x)'--);
```

and thus this axiom is the key to the formal semantics of protocols.

The proof proceeds by calling upon appropriate axioms mentioned in this section and relevant inference rules of the logic. This and another example are treated in depth in Volume III, the library of models.

Chapter 4

Theories of Availability

In this chapter we present a collection of formal properties which are meant to formalize various notions of *availability* (also known as *service assurance* or *denial of service*) for computer systems. We have not tried to cover all possible meanings of the term “availability”. We have focussed on two kinds of availability: (1) requirements that services are provided in a timely manner, which implicitly includes requirements on efficient allocation of resources in general, and (2) fault tolerance. We have also examined the general topic of making policies dynamic so as to facilitate tradeoffs with other requirements and reconfiguration to assure service.

In section 4.1 we give some simple requirements on theories of availability, including some informal meanings and some threats to availability that we want our theories to capture. In section 4.2 we describe some techniques for ensuring some of the meanings of availability that we want our formal theories to deal with. In section 4.3 we describe the way we will formally represent systems with availability requirements as mathematical objects. In section 4.4 we present the formal properties we have developed to capture various aspects of availability. In section 4.5, we give one of the semiformal examples we have formulated to help drive the development of our formal theories.

4.1 Requirements of Availability Theories

In order for our theories of availability to be reasonable, they must meet two requirements. First, they must capture certain informal notions of availability, in the sense that systems that do not meet those informal notions of availability should fail to meet one of our theories, and conversely, that systems that satisfy those informal notions of availability should satisfy one of our theories. Second, our theories of availability must capture certain threats to service, in the sense that systems which do not include adequate countermeasures to those threats should fail to satisfy one of our theories.

The informal meanings of availability that we mean to capture with our theories include the following:

- The system will continue to function in the presence of faults.
- The system responds in a timely fashion.
- The system can reconfigure itself to optimize response.

The threats to service we mean to capture with our theories include the following:

- hardware and software faults
- resource competition, both from legitimate competitors and malicious processes (e.g. “worms”)
- unpredictable scheduling algorithms
- thrashing

(These threats fall naturally into two groups: (1) not having enough resources, due to excessive demand or faults or both, and (2) having enough resources but not managing them correctly).

4.2 Techniques for Ensuring Availability

In this section we describe some of the techniques for ensuring service that we want our formal theories to deal with, in the sense that some of our formal theories may be satisfied by the proper use of these techniques.

4.2.1 Replication

One way to tolerate parts of a system becoming faulty is by replicating the parts that can fail. The system must then manage the replicated elements correctly to achieve fault tolerance. Voting algorithms can be used to combine the reports of replicated sensors into fault tolerant reports. Voting is also used to resolve conflicting actions directed by replicated processors, some of which may be faulty. Various kinds of agreement protocol are used to ensure that distributed communication does not create an erroneous impression that a system is faulty when it is not (e.g. by causing inconsistencies in data due to different interleavings of atomic parts of distributed transactions. Such inconsistencies will be treated as faults by voting algorithms). We want our theories of availability to be able to take account of the assurance provided by replication, while at the same time being sensitive to the possibility that replication is not managed correctly.

4.2.2 Static and Dynamic Resource Allocation

Resource allocation problems can sometimes be handled by static allocation and scheduling of resources. In this approach, resources are allocated according to a fixed cyclic schedule, so that it is completely predictable at any given time what processes, tasks, or users will have access to which resources. A corollary to this is that mechanisms for demand-driven service requests, such as hardware interrupts, must be excluded. When using static scheduling, I/O and service requests must be handled by polling. Utilization of resources can fluctuate even with static scheduling, but the *allocation* of resources (that is, ceilings on resource usage by various entities) must be static.

Static resource allocation has the advantage of being very predictable and very tractable to analyze rigorously. It has the disadvantage that it

does not optimize resource usage. There are some techniques known for improving on static resource allocation [32], but for applications with critical availability requirements, the state of the art in scheduling does not support any great degree of dynamic resource allocation. Most schemes for dynamic resource allocation cannot be shown to yield predictable behavior in arbitrary situations. We want our availability theories to be capable of modeling both static and dynamic resource allocation strategies.

4.3 System Representations for Availability

4.3.1 State Machine Representation

The way we will represent systems with critical availability requirements is based on the representations of systems used in the restrictiveness model, namely, state machines that interact with their environment by exchanging events. The relevant definitions are given in section 2.1.2.1 and section 2.1.3.

4.3.2 Representing Timing Properties

The principal extension we make to the state machine framework for purposes of availability analysis is to incorporate timing information about the machine's execution into the model. This is relevant to availability for two reasons. First, we need timing information to be able to state timeliness requirements. Second, we need to have the passage of time represented in order to be able to state dynamic versions of policies like restrictiveness that can adapt to system reconfiguration.

In this section we describe how we will incorporate timing information into the state machine formalism. In the project proposal, we said that we would use synchronous deterministic security as the basis for representing systems with availability properties, because synchronous deterministic security has the notion of hard real time built into it. We have developed ways to incorporate timing information into the restrictiveness formalism which are adequate for stating the availability properties we need to state. We have therefore decided to use extensions of the restrictiveness formalism in-

stead of synchronous deterministic security. There are several reasons why this is desirable. It is better to have one formalism for modeling systems in Romulus for pedagogical reasons. By using restrictiveness, we can adapt existing tools and techniques instead of building entirely new ones for a new formalism. Finally the restrictiveness formalism is closer to commonly used formalisms for specifying systems like CSP and CCS.

We have developed two ways of incorporating timing information into the restrictiveness formalism. We will now describe each approach, contrast them, and then describe how we model users' observations of time in each of the approaches.

The first way of adding timing information to a state machine is by having special events which mark the passage of time, which we will call "ticks". Ticks are outputs of the machine. Time in this approach means "number of ticks". Inputs are of necessity treated as happening instantaneously, because an arbitrary number of inputs can happen between any two ticks by input totality. There can be more than one kind of tick associated with a system, and they can have different levels. For example, a system can express the smallest granularity of time by one kind of tick, and coarser granularities by other kinds of tick, with many of the first kind of tick occurring between two successive ticks of the second kind. The finest granularity "tick" can be used solely for specifying timing properties, and can have level "system-high" if security is being specified and we don't want to assume that users have arbitrarily good clocks. Ticks corresponding to users' clocks can have level "system-low".

The second way of adding timing information to a state machine is by having a clock or clocks as part of the state. In this approach, the amount of time that an input, output, or internal transition takes is specified by the transition relation. Inputs need not be regarded as instantaneous, because the transition relation can force the response to an input to include incrementing the clock. Multiple clocks can be used for the same purpose as multiple kinds of tick in the previous approach, that is, to keep track of different granularities of time.

The choice of approach depends on a variety of factors, most of which are application specific. One general comment which can be made contrasting the two approaches is that the "tick" approach lends itself to specifying

“soft” timing properties, whereas the “state clock” approach lends itself to specifying “hard” timing properties. For example, ticks can easily be used to add time to a state machine without specifying anything about the rate at which things happen, just by specifying the machine so that it allows a tick to happen at any time, nondeterministically. The state clock approach is useful for hard time because it allows us to specify timing properties for inputs as well as outputs.

How do we model the observation of time by users? In the “tick” approach, observation of time is just observation of the occurrences of ticks. This is the normal notion of “observation” from the original, timeless restrictiveness model. As mentioned above, different ticks with different security levels can be used to represent different granularities of time, with observability being controlled by security levels. In the “state clock” approach, we can model observation of the clocks by requiring that various projection functions contain the values of various clocks. Observability of a given clock can be controlled by including it in the projection functions for some levels, and not others.

4.4 Formal Availability Properties

In this section we describe the formal theories we have developed to express our informal notions of availability. In section 4.4.1 we describe a general theory of fault tolerance. In section 4.4.2 we describe a general approach to stating timeliness requirements. In section 4.4.3, we describe dynamic versions of deducibility security and restrictiveness.

4.4.1 t -Fault Tolerance

In this section we describe the theory of what is called *t-fault tolerance* in the fault tolerance literature. Informally, a system is *t-fault tolerant* if it is the case that the system will continue functioning normally as long as there are at most t faults. This is the informal notion we wish to formalize here.

First of all, we model the notion of a fault in the state machine framework as a particular class of inputs called *fault events*. These are the actual

events of a fault occurring. For example, a sensor beginning to act erratically would be modeled as the occurrence of a fault event. The kinds of faults allowed, and the effects of those faults on system performance, are part of the description of the state transition function (see section 4.5.1 for examples).

The inputs and outputs which are not fault events are referred to as the *fault tolerant events* because they should occur correctly, even in the presence of faults, as long as there are fewer than t faults, and the system's fault tolerance is correctly implemented. For example, the report of a sensor value to a user which is generated by combining the reports of redundant sensors would be a fault tolerant event.

The parameter t is incorporated into the analysis by specifying the transition relation of the system so that fault events do not change the state of the system after there have been t of them. Thus, the system is defined in such a way that only the first t faults have any effect. This effectively means that in any trace of the state machine, there are only t “real” faults. Given this, the fault tolerance requirement is reduced to the requirement that the state machine behave “normally” in all possible traces.

How do we formalize the notion of the system behaving “normally” in the presence of faults? We have developed an approach to formalizing this notion based on restrictiveness. We imagine that we are trying to prove security for the system, with two security levels: *fault* and *nonfault*, where *nonfault* < *fault*. The fault events are assigned level *fault* and the fault tolerant events are assigned level *nonfault*. The fault tolerance policy for the system is then just restrictiveness for this assignment of “levels”. If we unpack the definition of restrictiveness, we arrive at the following definition:

Definition 3: Given a state machine and a division of its events into fault events and fault tolerant events, the state machine *satisfies its fault tolerance policy* (or simple *is fault tolerant*) if and only if the following condition is satisfied: there exists a function p (called the *fault tolerant projection*) whose domain is the set of states of the machine, such that:

1. For every reachable state s and every fault event f , if the state machine can make a transition from s to s' accompanied by f , then $p(s) = p(s')$ (“Fault events do not affect the fault tolerant projection”).
2. For every pair of reachable states s and t and every fault tolerant input

i , if $p(s) = p(t)$ and the machine can make a transition from s to s' accompanied by i , then there exists a state t' such that the machine can make a transition from t to t' accompanied by i , and $p(t') = p(s')$ ("The effect of fault tolerant inputs on the fault tolerant projection depends only on the projection").

3. For every pair of reachable states s and t and every output o , if $p(s) = p(t)$ and the machine can make a transition from s to s' accompanied by o , then there exists a state t' such that the machine can make a sequence of transitions, beginning at t and ending at t' , with one of the transitions in the sequence accompanied by o , and all the others internal, and such that $p(t') = p(s')$ ("The fault tolerant outputs depend only on the fault tolerant projection").
4. For every pair of reachable states s and t , if $p(s) = p(t)$ and the machine can make an internal transition from s to s' , then there exists a state t' such that the machine can make a sequence of internal transitions beginning at t and ending at t' , and $p(t') = p(s')$ ("Fault tolerant internal processing depends only on the fault tolerant projection").

□

The fault tolerant projection is meant to be a function which takes a state of the machine, possibly including some faulty components, and returns a kind of "sanitized version" of the state in which the effects of faults have been removed. The clauses of the above definition of fault tolerance ensure two things: (1) that the value of the fault tolerant projection can only reflect nonfaulty inputs and processing (i.e. that it really is a "sanitized version" of the state), and that (2) all observable behavior that is supposed to be immune to faults depends only on the fault tolerant projection (and so is not influenced by faults). The choice of the fault tolerant projection implicitly defines what the "normal" or "nonfaulty" behavior of the system is. The above definition essentially states necessary and sufficient conditions on what we define to be "normal" behavior for the behavior in the presence of faults to be "normal".

The basic idea of formalizing fault tolerance as a form of security or noninterference property was originally put forth in [61]. The theory of fault

tolerance presented in this section is essentially a formal working-out of the details of the idea in [61].

4.4.2 Timeliness Properties

Many availability requirements are simple timeliness requirements. These timeliness requirements can be “hard time” requirements, by which we mean requirements that things happen within certain bounds on elapsed time, or “soft time” requirements, by which we mean requirements that things happen within *some* elapsed time. A requirement that a system provide certain services periodically is essentially a requirement that the system either “do something” at least once every T seconds (the hard time case), or “do something” infinitely often (the soft time case). A requirement that the system provide a certain service on demand is essentially a requirement that the system either “do something” within T seconds of receiving some request (the hard time case), or “do something” eventually after receiving some request (the soft time case).

What about these kinds of requirement should be formalized generically? We have spent some time attempting to formalize the general notions of “service” and “request”, but this seems to be relatively unbeneficial because (1) what constitutes a “service” or a “request” is highly specific to the particular application being modeled, and so is difficult to generalize without making it trivial, and (2) the major problem in satisfying getting these kinds of property does not seem to be errors in writing down precisely what is meant by “service” and “request”, so formalizing these notions will not address the real problems of meeting timeliness requirements. The real problems associated with timeliness requirements are classical scheduling problems, e.g. how can system resources be allocated to tasks in such a way that all deadlines are met? These problems are being addressed by a large community of people, so it would not be profitable or appropriate for the Romulus project to attempt to solve these problems. What *is* appropriate is to provide formalisms capable of (1) formalizing the timeliness requirements on systems, (2) expressing the solutions that workers in the field come up with, and (3) proving that the solutions satisfy the requirements.

The “tick” and “state clock” approaches to incorporating timing behavior

into a state machine are very general approaches which should be general enough to express various scheduling solutions. The question we will address in the remainder of this section is how we formalize timeliness requirements on systems.

The answer to this is actually quite simple. Timeliness requirements, either hard or soft, can be formalized simply as statements about the set of complete traces of a system which incorporates time by the "tick" or "state clock" approach. In fact, most timeliness requirements can be stated in the form "every complete trace of the system has property P ". We can go further and say that most of the properties P are statements about finite traces within C . For example, a typical soft time scheduling requirement would be the requirement that a schedule be *fair*, which would be formalized by an assertion like

Every complete trace C has the property that there is no process p which, at some point in C , becomes ready to run and remains ready to run for the rest of C , but is never run.

The system about which this statement is made would need to have some components of state giving the set of processes which are ready and the set of processes which are currently running.

A hard time fairness requirement would be formalized by an assertion like

Every complete trace C has the property that there is no process p which, at some point in C , becomes ready to run and remains ready to run for t seconds without being run.

The system about which this statement is made would need to have the same components of state as the system with the soft time requirement, but with hard time incorporated into the state.

In short, if we incorporate timing information into state machines, it is relatively straightforward to state timeliness properties as statements about complete traces. Further, it is often possible to transform such requirements from statements about complete traces to state invariants, particularly in the hard time case. For example, consider the hard time fairness assumption above. If we add "history variables" to the state of the machine which

record the last time at which a process became ready to run, then the above statement on complete traces reduces to the statement

In every reachable state, there are no processes which became ready to run more than t seconds before the current time which are not running now.

Since this is a universal quantification over reachable states, it is a state invariant, and so may be susceptible to proof by induction.

The Romulus library of models, Volume III, contains a model that further explores real-time systems.

4.4.3 Dynamic Security Properties

4.4.3.1 Need for Dynamic Security Properties

Current theories of security like deducibility security and restrictiveness model systems as state machines with certain *security parameters* “attached” to them. The state machines themselves (the states, the events, the states that the machine can start in, the state transitions that it can make) just describe how the system behaves. In terms of information, the state machines themselves only describe how information flows through the system. One cannot determine from looking at the state machine alone whether the system it defines is secure or not. To determine whether the system is secure, one needs to examine the security parameters. In the case of restrictiveness, the security parameters are the assignments of security levels to the inputs and outputs of the machine. The security parameters define the sensitivities of the various kinds of information that pass through the system. Once these sensitivities are defined, the information flows which can be found from the state machines alone can be labeled secure or insecure, depending on whether the information is flowing from lesser to greater security levels or not.

Existing models are *static* in the sense that the security parameters of the system (the assignments of security levels to inputs and outputs) cannot change. There are several reasons why it is desirable to have the security parameters be *dynamic*, that is, to have it be possible for events to have

different levels at different times, and to be able to change the classification which an event is regarded as being. First, the same physical events will often have different levels at different times, e.g. inputs on a given physical interface during different login sessions. Second, tradeoffs between security and other properties may require that information is leaked; dynamic security properties can model such leakage as a downgrade, and specify exactly what is allowed to be downgraded, and how fast. Finally, system reconfiguration to meet changing conditions may require changes in security policy.

The approach of regarding leakage as downgrading is a relatively untried alternative to bandwidth analysis (although it is suggested by the work in [60]). In this approach, instead of specifying a system which is not restrictive and then trying to determine how much leakage there is, one specifies the leakage and then tries to prove that the specified allowable leakage is all the leakage. It seems possible that it would be easier to prove the absence of any channels except those specifically allowed and specified than it would be to find the channels in a specification which was just written to express certain functionality. It also seems a better approach to security engineering, in that it forces the designer to make decisions about what security "slippage" is allowed, and to design that slippage in, with the designer in control, than to just "let the chips fall where they may".

We will first describe how we make deducibility security for state machines dynamic. We will then describe how dynamic deducibility security is strengthened to get a dynamic version of the restrictiveness model. A dynamic version of deducibility security was previously formulated in [59], but the version presented below is much simpler, and also more general.

4.4.3.2 Dynamic Deducibility Security

The idea for making deducibility security dynamic is very simple, and can be stated at the level of possible worlds. We will first state it this way, and then say how it is instantiated to state machines.

The original instantiation of deducibility security had one pair of information functions for each security level l . One information function represented the information possessed by users at level l in a given world (the *low view*), and the other represented the information which was not supposed to be

inferable by users at level l in a given world (the *hidden information*). This definition of security can be made dynamic merely by having two information functions for each security level l and each time t (where time is represented by some set of values such as the integers or real numbers). One information function represents the information which is possessed by users at level l at time t in a given world (the *low view at time t*), and the other represents the information which is not supposed to be inferable by users at level l at time t in a given world (the *hidden information at time t*).

How would this approach be used to model things like downgrading information? Suppose some input is SECRET when it occurs, and is later downgraded to UNCLASSIFIED. Suppose the input occurs at time t_1 , and is downgraded at time t_2 . For times t such that $t_1 < t < t_2$, the input would be part of the hidden information at time t for level UNCLASSIFIED, and would not be part of the UNCLASSIFIED view at time t . For times $t \geq t_2$, the input would be part of the UNCLASSIFIED view at time t , and would not be part of the hidden information at time t for level UNCLASSIFIED. Since there are separate hidden and view functions for each time, events can be in the view for a given level at some times and not others, and likewise for the hidden information.

How do we instantiate dynamic deducibility for possible worlds to the state machine framework? The basic definition of "state machine" is unchanged, but the security parameters must be generalized. Instead of a fixed mapping from inputs and outputs to security levels, the security parameters associated with a state machine are expressed as a function which takes a finite trace t and a particular occurrence e of an input or output in t and returns a security level. Thus, a particular input or output may have one level in a trace t , and another level in a trace t' extending t .

Given such a security structure, we can now define the view and hidden information function for a given security level l and time t . The possible worlds are now *complete* traces of the state machine. The view of a complete trace C at level l and time t is the sequence of inputs and outputs in C which (1) occur at or before time t in the trace, and (2) have level $\leq l$ in the trace at time t . The hidden information for level l and time t is the sequence of inputs in C which (1) occur at or before time t in the trace, and (2) do *not* have level $\leq l$ at time t in the trace.

The above definition requires that there be some notion of time defined for the state machine being used. This notion of time can be any of those discussed in section 4.3.2.

4.4.3.3 Dynamic Restrictiveness

In this section we describe a version of the restrictiveness model which uses dynamic security parameters. Like dynamic deducibility security, we obtain dynamic restrictiveness simply by taking something which formerly only depended on a security level and make it depend on a security level and a time. The “something” in question now is the projection function. The static restrictiveness model requires that there exist a projection function p_l for each security level l satisfying certain properties. For dynamic restrictiveness, we require that there exist a projection function $p_{l,t}$ for each security level l and each time t which satisfies a similar set of properties.

We will now give the formal definition of dynamic restrictiveness. The reader who is familiar with ordinary restrictiveness will recognize the clauses of the definition as a straightforward translation of the clauses of ordinary restrictiveness with the following replacements:

- “Reachable state” is replaced by “state reachable by time t ”.
- “ $\leq l$ ” is replaced by “ $\leq l$ at time t ”.
- “Can make a transition” and “Can make a sequence of transitions” replaced by “Can make a transition by time t ” and “Can make a sequence of transitions by time t ” respectively.

Definition 4: Given a state machine with an associated notion of time, and dynamic security parameters as described in section 4.4.3.2, the machine is *dynamically restrictive* if and only if there exists a function $P_{l,t}$ for each security level l and each time t (called the *level l projection at time t*) which satisfies the following conditions:

1. For every state s reachable by time t and every input i which is not $\leq l$ at time t , if the state machine can make a transition from s to s' accompanied by i by time t , then $p_{l,t}(s) = p_{l,t}(s')$.

2. For every pair of states s and r reachable by time t and every input i which is $\leq l$ at time t , if $p_{l,t}(s) = p_{l,t}(r)$ and the machine can make a transition from s to s' accompanied by i by time t , then there exists a state r' such that the machine can make a transition from r to r' accompanied by i by time t , and $p_{l,t}(r') = p_{l,t}(s')$.
3. For every pair of states s and r reachable by time t and every output o which is $\leq l$ at time t , if $p_{l,t}(s) = p_{l,t}(r)$ and the machine can make a transition from s to s' accompanied by o by time t , then there exists a state r' such that the machine can make a sequence of transitions by time t , beginning at r and ending at r' , with one of the transitions in the sequence accompanied by o , and all the others either internal or accompanied by an output o' which is not $\leq l$ at time t , and such that $p_{l,t}(r') = p_{l,t}(s')$.
4. For every pair of states s and r reachable by time t , if $p_{l,t}(s) = p_{l,t}(r)$ and the machine can make a transition from s to s' by time t which is either internal or accompanied by an output which is not $\leq l$ at time t , then there exists a state r' such that the machine can make a sequence of transitions by time t , beginning at r and ending at r' , with each transition in the sequence either internal or accompanied by an output which is not $\leq l$ at time t , and such that $p_{l,t}(r') = p_{l,t}(s')$.

□

4.5 Example

In this section we describe one of the examples we've looked at to drive the formulation of our theories of availability. This example is described semi-formally. Other examples can be found in Volume III, the library of models.

4.5.1 Fault Tolerant Sensor

In this section we describe two versions of a simple fault tolerant sensor system as state machines, state their fault tolerance requirements using the

theory of fault tolerance described in section 4.4.1, and derive necessary and sufficient conditions on the degree of replication and the fault tolerance algorithm for the two systems to be able to withstand t faults. The conditions we derive agree with the classical conditions from the fault tolerance literature.

The difference between the two versions of the sensor system we will consider is in the kinds of fault that are allowed to occur (commonly referred to as the *failure modes* or *failure semantics* of the systems). In the first version of the system, a faulty sensor will be assumed to simply become inert, that is, it will stop reporting values for what it is measuring. Such faults are commonly called *crash failures* in the literature on fault tolerance. In the second version of the system, a faulty sensor will be permitted to begin reporting arbitrary values for the quantity it is measuring. Such faults are commonly called *Byzantine faults* in the literature on fault tolerance.¹

4.5.1.1 Fault Tolerant Sensor with Crash Faults

We will first describe the system informally. The fault tolerant sensor system has n copies of a given kind of sensor, all measuring the same quantity. At any time, a sensor may become faulty. Nonfaulty sensors report their values to a voting algorithm. The voting algorithm will report a given value for the quantity being monitored if it gets r or more votes for that value, where r is some value greater than 0.

We model this situation as a state machine as follows. The sensors report measurements from some set M . The collection of sensors is a set S with n elements in it. The state of the machine consists of (1) a function from S into M (the values currently reported by the sensors, if any), (2) a boolean value saying whether the last value received by the sensors has been reported yet, and (3) a subset of S (the sensors which have become faulty). We will use the notation $\langle V, b, F \rangle$ for the state in which V is the function giving the values of the sensors, b is the boolean saying whether the last value has been reported, and F is the set of faulty sensors.

There are two kinds of input: (1) for each $s \in S$, there is an input *fault* _{s} ,

¹This phrase derives from the behavior of certain generals of the Byzantine Empire who, while allegedly working together on the same side, would commonly betray each other.

which represents sensor s becoming faulty, and (2) for each value $m \in M$, there is an input in_m which represents the sensor ensemble receiving value m from the environment. There is one kind of output: for each $m \in M$, there is an output out_m which represents the voter reporting value m .

The initial states of the machine are any state in which the set of faulty sensors is the empty set (all sensors are assumed to be working correctly initially), the boolean value indicated that the last value has been reported, and all sensors are reporting the same value.

If the machine is in a state $\langle V, b, F \rangle$, and $fault_s$ occurs, the machine will remain in the same state if F contains t or more sensors, and will otherwise make a transition to the state $\langle V, b, F \cup \{s\} \rangle$. If in_m occurs, the machine will make a transition to some state $\langle V', \text{false}, F \rangle$ where $V'(s) = m$ for all $s \notin F$ (in other words, all nonfaulty sensors will report the correct value, and the faulty sensors will do something arbitrary).

If the machine is in a state $\langle V, b, F \rangle$, it can make a transition to $\langle V, \text{true}, F \rangle$ with output out_m if and only if b is “false” (that is, the last sensor value received has not yet been reported), and there are at least r sensors s not in F such that $V(s) = m$. (It is in this part of the definition that the failure semantics is expressed: since faulty sensors do not report a value, the voter will produce an output if sufficiently many of the *nonfault* sensors report that value). These are all the possible transitions of the machine.

Note that it is an invariant of this state machine that the nonfaulty sensors all report the same value. We will now describe the fault tolerant projection of the state, as described in section 4.4.1. The fault tolerant projection takes a reachable state $\langle V, b, F \rangle$ and returns the pair $\langle m, b \rangle$ where m is the value reported by the nonfaulty sensors (if any), and some fixed default value m_0 otherwise. This definition follows that intuitive guidelines described in section 4.4.1 for the fault tolerant projection: it picks out the state information residing in the nonfaulty components of the machine. The fault events are, of course, the events of the form $fault_s$.

We will now analyze what relationships would have to hold between the values of n , t , and r in order for the above machine to satisfy the fault tolerance policy associated with the above fault tolerant projection and collection of fault events. To do this, we will go through the clauses of the fault tolerance policy.

Fault events do not affect the fault tolerant projection. Since occurrences of fault events do not affect the boolean b or the function V , if there are nonfaulty sensors remaining after the occurrence of an event $fault_s$, then there must have been nonfaulty sensors before that event, and they must all be reporting the same value after as they did before. If there were no nonfaulty sensors before the event, then there will be none after, and again the fault tolerant projection will not change. Thus, the only way that the value of the fault tolerant projection could change with the occurrence of an event $fault_s$ is if there were nonfaulty sensors before the event, and none after. Such a transition is reachable if t (the maximum number of faults that the sensor system will respond to) is $\geq n$ (the number of replicated sensors there are). If $t < n$, then there will be at least one nonfaulty sensor in every reachable state, and so there will be no reachable transition in which a fault event affects the fault tolerant projection. Thus, for the system to satisfy its fault tolerance property, it must be the case that $t < n$.

The effect of fault tolerant inputs on the fault tolerant projection depends only on the projection. Given that the requirement from the previous clause, that $t < n$, this clause will be satisfied, because if s_1 and s_2 are reachable states which have the same fault tolerant projections, then (if $t < n$) both s_1 and s_2 will have nonfaulty sensors. The effect of a fault tolerant input in_m will be to make the nonfaulty sensors of both s_1 and s_2 register the value m , and to make the boolean value in the state "false", so the states after the transitions will have the same fault tolerant projections.

The fault tolerant outputs depend only on the fault tolerant projection. Suppose $\langle V_1, b_1, F_1 \rangle$ and $\langle V_2, b_2, F_1 \rangle$ are reachable states with the same fault tolerant projection, and the machine can make a transition from state $\langle V_1, b_1, F_1 \rangle$ to state $\langle V'_1, b'_1, F'_1 \rangle$ accompanied by output out_m . For the system to satisfy its fault tolerance policy, it is necessary that there exist a state $\langle V'_2, b'_2, F'_2 \rangle$ with the same fault tolerant projection as $\langle V_2, b_2, F_2 \rangle$ and such that the machine can make a transition from $\langle V_2, b_2, F_2 \rangle$ to $\langle V'_2, b'_2, F'_2 \rangle$ accompanied by output out_m . We will now consider what values of n , t , and r make this true.

First, since $\langle V_1, b_1, F_1 \rangle$ can make an output, b_1 must be "false". Since b_1 is part of the fault tolerant projection, b_2 must be "false" too. By definition of the transition relation, $V'_1 = V_1$, $b'_1 = \text{true}$, and $F'_1 = F_1$. There must be at

least r nonfaulty sensors in $\langle V_1, b_1, F_1 \rangle$ reporting value m . Since $\langle V_1, b_1, F_1 \rangle$ is reachable, all nonfaulty sensors must be reporting value m . Since $\langle V_2, b_2, F_2 \rangle$ is reachable and has the same fault tolerant projection, all nonfaulty sensors in $\langle V_2, b_2, F_2 \rangle$ must also be reporting value m . If there are at least r nonfaulty sensors in $\langle V_2, b_2, F_2 \rangle$, then the machine can make a transition to $\langle V_2, \text{true}, F_2 \rangle$ accompanied by out_m , and $\langle V_2, \text{true}, F_2 \rangle$ will have the same fault tolerant projection as $\langle V_1, \text{true}, F_1 \rangle$. Thus, the fault tolerance policy will be satisfied if, in the above circumstances, there will necessarily be at least r nonfaulty sensors in $\langle V_2, b_2, F_2 \rangle$. This could fail if it is possible to have enough faults to leave fewer than r nonfaulty sensors, that is, if $n - t < r$. Thus, in order to satisfy this clause of the fault tolerance policy, it is necessary the $n - t \geq r$.

The above system will then satisfy its fault tolerance policy with crash fault failures if and only iff $t < n$ and $n - t \geq r$. Assuming a fixed t , this means that n must be at least $t + 1$, and r can be at most $n - t$. The first condition is the classical one from the fault tolerance literature: a voting algorithm needs one more replicated element than the number of faults in order to tolerate crash failures. The second condition is really a condition on the number of votes required by the voting algorithm. The voting algorithm must not require too many votes (at most $n - t$) before it will report a value. In fact, since $n - t \geq 1$, this requirement can be satisfied by letting r equal 1. This will ensure that the voting algorithm does not require too many votes whenever there is enough replication to overcome the required number of faults. The reason the voting algorithm only needs to get one vote for a value before it reports the value is because the failures are assumed to be crash failures; the voter will never get an incorrect vote.

4.5.1.2 Fault Tolerant Sensor with Byzantine Faults

Describing the fault tolerant sensor system with Byzantine failure semantics as a state machine requires only one change to the description above, namely, it will now be the case that the machine will make a transition from $\langle V, \text{false}, F \rangle$ to $\langle V, \text{true}, F \rangle$ accompanied by output out_m if there are at least r sensors, *faulty or not*, which are reporting the value m . This reflects the fact that with Byzantine faults, faulty sensors are allowed to report faulty values.

We will now go through the clauses of the fault tolerance policy and see what new conditions on n , t , and r are necessary to satisfy the policy with Byzantine faults.

Fault events do not affect the fault tolerant projection. The analysis here is the same as that in the crash failure case. This clause will be satisfied if $t < n$.

The effect of fault tolerant inputs on the fault tolerant projection depends only on the projection. Again, this will be satisfied if $t < n$, by the same argument as above.

The fault tolerant outputs depend only on the fault tolerant projection. The analysis of this clause is different than in the crash failure case. Suppose $\langle V_1, b_1, F_1 \rangle$ and $\langle V_2, b_2, F_1 \rangle$ are reachable states with the same fault tolerant projection, and the machine can make a transition from state $\langle V_1, b_1, F_1 \rangle$ to state $\langle V'_1, b'_1, F'_1 \rangle$ accompanied by output out_m . For the system to satisfy its fault tolerance policy, it is necessary that there exist a state $\langle V'_2, b'_2, F'_2 \rangle$ with the same fault tolerant projection as $\langle V_2, b_2, F_2 \rangle$ and such that the machine can make a transition from $\langle V_2, b_2, F_2 \rangle$ to $\langle V'_2, b'_2, F'_2 \rangle$ accompanied by output out_m . We will now consider what values of n , t , and r make this true in the Byzantine failure case.

First, since $\langle V_1, b_1, F_1 \rangle$ can make an output, b_1 must be "false". Since b_1 is part of the fault tolerant projection, b_2 must be "false" too. By definition of the transition relation, $V'_1 = V_1$, $b'_1 = \text{true}$, and $F'_1 = F_1$.

If there are r or more faulty sensors, then it could be that (1) $\langle V_1, b_1, F_1 \rangle$ is a state in which the nonfaulty sensors are reporting a value $m' \neq m$, (2) at least r faulty sensors are reporting the value m , and (3) $\langle V_2, b_2, F_2 \rangle$ is a state in which all sensors are reporting the value m' . Such states (1) are reachable, (2) have the same fault tolerant projection, and (3) $\langle V_1, b_1, F_1 \rangle$ can make a transition with output out_m , but $\langle V_2, b_2, F_2 \rangle$ cannot make such a transition. In order to rule out this possibility, it must be the case that there are less than r faulty sensors in any reachable state. The only way for this to be true is if the largest number of faulty sensors that there can be in any reachable state is less than r . The largest number of faulty sensors there can be is t , so it must be that $t < r$.

If there are fewer than r nonfaulty sensors in $\langle V_1, b_1, F_1 \rangle$, then it could

be that $F_2 = F_1$ and V_2 is the function which maps a sensor s to $V_1(s)$ if s is nonfaulty, and maps it to some value $m' \neq m$ otherwise. Such a state is reachable, and has the same fault tolerant projection as $\langle V_1, b_1, F_1 \rangle$, but cannot make a transition to another state accompanied by out_m . Thus, in order to satisfy the fault tolerance property, it must be that there are at least r nonfaulty sensors in every reachable state. The only way for this to be true is if the smallest number of nonfaulty sensors that there can be in any reachable state is at least r . The smallest number of nonfaulty sensors there can be is $n - t$, so it must be that $n - t \geq r$.

If the above three conditions ($t < n$, $t < r$, and $n - t \geq r$) are satisfied, then the fault tolerance policy for the system with Byzantine failures is satisfied. We will now give the semiformal argument for this.

By the three conditions, there must be at least r nonfaulty sensors, and fewer than r faulty sensors, in both $\langle V_1, b_1, F_1 \rangle$ and $\langle V_2, b_2, F_2 \rangle$ (although the collections of faulty sensors in the two states need not be the same). Thus, in both of these states, the measurement reported by the nonfaulty sensors is reported by at least r sensors, and any different values which are reported by the nonfaulty sensors are reported by fewer than r sensors. Thus, m must be the value reported by the nonfaulty sensors in $\langle V_1, b_1, F_1 \rangle$. Because $\langle V_2, b_2, F_2 \rangle$ has the same fault tolerant projection, all the nonfaulty sensors in this state must be reporting the value m as well, so there are at least r sensors reporting the value m . By the definition of the transition relation, the machine can make a transition from $\langle V_2, b_2, F_2 \rangle$ to $\langle V_2, \text{true}, F_2 \rangle$, which will have the same fault tolerant projection as $\langle V_1', b_1', F_1' \rangle$.

Assuming a fixed t , the above conditions require that $n \geq 2t + 1$ and $t + 1 \leq r \leq n - t$. The first condition is the classical one from the fault tolerance literature: a voting algorithm needs one more replicated element than twice the number of faults in order to tolerate Byzantine failures. The second condition requires that the voter requires neither too many *nor* too few votes to report a value. In the crash failure case, no number of votes greater than 0 was too few, because all votes were reliable. In fact, since $n - t \geq t + 1$, this requirement can be satisfied by letting r be $t + 1$. This will ensure that the voting algorithm requires enough votes to only get that many votes for a correct value, and does not require too many whenever there is enough replication to overcome the required number of faults.

The fact that our restrictiveness-based theory of fault tolerance allows us to derive the classical results in fault tolerance is fairly good evidence that it is a correct generic formulation of fault tolerance requirements. Further, since it is a version of restrictiveness, we can use the present Romulus Security Condition Generator to reduce the proof of a fault-tolerance property to simple statements about inputs and outputs.

Appendix A

Probability

A probability measure is a function that takes a set of possible worlds and returns the probability that an execution history of the system will be in that set. Probability measures do not usually assign probabilities to every set of possible worlds, but only to some. Thus, the description of the probability measure must include what sets are assigned a probability, and what probability those sets are assigned.

We will now describe how a probability measure on the set of complete traces of a probabilistic state machine is computed from a probabilistic transition relation for the machine. Given a finite trace of such a machine, we can compute a probability (called the *weight*) associated with the trace by taking the probabilities of each of the transitions occurring in the trace and multiplying them together. Using this, we can compute a value for an arbitrary set S of complete traces (called the *outer measure* of S and denoted by $\mu_o(S)$) as follows: for any integer n , let S_n be the set of truncations of elements of S to finite traces of length n . We can then compute the weights of all the elements of S_n and add them up. Call this number $\mu_{o,n}(S)$. It is easy to show that $n > m$ implies that $\mu_{o,n}(S) < \mu_{o,m}(S)$. Define $\mu_o(S)$ to be the greatest lower bound of all the $\mu_{o,n}$'s.

Denote the set of all complete traces *not* in a set S by $-S$. We say a set S of complete traces is *measurable* if and only if

$$\mu_o(S) + \mu_o(-S) = 1$$

The probability associated with a measurable set S is $\mu_o(S)$.

It is beyond the scope of this document to prove that this function satisfies the axioms for a probability measure, so we will omit the proof.

Bibliography

- [1] J. Alves-Foss and K. Levitt. A model of event systems in higher order logic: Sequence and event system theories. Technical Report CSE-90-45, Division of Computer Science, University of California, Davis, November 1990.
- [2] J. Alves-Foss and K. Levitt. A security property in higher order logic: Restrictiveness and hook-up theories. Technical Report CSE-90-46, Division of Computer Science, University of California, Davis, December 1990.
- [3] J. Alves-Foss and K. Levitt. Mechanical verification of secure distributed systems in higher order logic. In *Proceedings of the 1991 International Meeting on Higher Order Logic Theorem Proving and its Applications*, pages 263–278, 1991.
- [4] J. Alves-Foss and K. Levitt. Verification of secure distributed systems in higher order logic: A modular approach using generic components. In *Proceedings of the Symposium on Security and Privacy*, pages 122–135, Oakland, CA, 1991. IEEE.
- [5] R.K. Bauer, T.A. Berson, and R.J. Feiertag. A key distribution protocol using event markers. *ACM Transactions on Computer Systems*, 1(3):249–255, August 1983.
- [6] D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford MA, 1976.

- [7] S. Brackin and S-K Chin. Server-process restrictiveness in HOL. In *HOL User's Group Workshop*, Vancouver, Canada, August 1993. Springer Verlag.
- [8] S. H. Brackin and S-K Chin. Technical report. (in progress).
- [9] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1), February 1990.
- [10] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the hol theorem prover. Technical Report 265, University of Cambridge, Computer Laboratory, August 1992.
- [11] D.E.Denning and G.M. Sacco. Timestamps in key distribution protocols. *CACM*, 24(8):533-536, August 1981.
- [12] Department of Defense. *Trusted Computer System Evaluation Criteria*, December 1985. DoD-5200.28-STD.
- [13] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644-654, November 1976.
- [14] Joseph A. Goguen and José Meseguer. Security policy and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11-20, Oakland, CA, April 1982. IEEE.
- [15] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *Proceedings of the Symposium on Security and Privacy*, pages 75-86, Oakland, CA, April 1984. IEEE.
- [16] Li Gong. Handling infeasible specifications of cryptographic protocols. In *Proceedings of The Computer Security Foundations Workshop IV*, pages 99-102, Franconia, NH, June 1991.
- [17] Li Gong and Geoffrey Hird. The ORA toolkit for analyzing cryptographic protocols: A user manual. (Draft), 1993.
- [18] Li Gong, Roger Needham, and Raphael Yahalom. A methodology for analyzing cryptographic protocols. In *Proceedings of 11th IEEE Symposium on Research in Security and Privacy*, pages 234-248, Oakland, CA, May 1990.

- [19] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, Englewood Cliffs, NJ, 1985.
- [20] Tanya Korelsky, David Rosenthal, and Daryl McCullough. Basic technology for sdi computer security: Security analysis using the romulus methodology. Technical Report RADC-TR-90-435 Vol I, Rome Air Development Center, Griffiss Air Force Base, December 1990. *
- [21] Daryl McCullough. Specifications for multilevel security and a hook-up property. In *Proceedings of the Symposium on Security and Privacy*, pages 161–166, Oakland, CA, April 1987. IEEE.
- [22] Daryl McCullough. Covert channels and degrees of insecurity. In *Proceedings of Computer Security Foundations Workshop*, pages 1–33, Franconia, NH, June 1988. The MITRE Corporation, M88-37.
- [23] Daryl McCullough. Foundations of Ulysses: The theory of security. Technical Report RADC-TR-87-222, Rome Air Development Center, May 1988.
- [24] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the Symposium on Security and Privacy*, pages 177–186, Oakland, CA, April 1988. IEEE.
- [25] Daryl McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6):563–568, June 1990.
- [26] J. A. McDermid and Qi Shi. Secure composition of systems. In *Proceedings of the Security Applications Conference*, pages 112–122. IEEE, 1992.
- [27] T. F. Melham. Automating recursive type definitions in higher order logic. Technical Report 146, University of Cambridge, Computer Laboratory, January 1989.
- [28] R. C. Merkle. Secrecy, authentication, and public key systems. In *UMI Research Press Michigan*, 1982.

*Although this report references the limited document noted above, no limited information has been extracted. USGO Agencies and their contractors; critical technology; Dec 90.

- [29] C.H. Meyer and M. Schilling. Secure program load with modification detection code. In *Proceedings of the 5th Worldwide Congress on Computer and Communication Security and Protection - SECURICOM 88, Paris*, pages 111–130, 1988.
- [30] Jonathan K. Millen. Hookup security for synchronous machines. In *Proceedings of the Computer Security Foundations Workshop III*, pages 84–90, Franconia, NH, June 1990. IEEE Computer Society Press.
- [31] R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1990.
- [32] Abha Moitra. Voluntary preemption: A tool in the design of hard real-time systems. Technical report, ORA, March 1989.
- [33] National Computer Security Center. *Integrity in Automated Information Systems*, September 1991.
- [34] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *CACM*, 21(12):993–999, December 1978.
- [35] R.M. Needham and M.D. Schroeder. Authentication revisited. *ACM Operating Systems Review*, 21(1):8–10, January 1987.
- [36] ORA. Romulus: A computer security properties modeling environment, final report - volume 1: Overview. Technical report, ORA, June 1990.
- [37] ORA. Romulus: A computer security properties modeling environment, final report - volume 2a and 2b: Theory of security and mathesis. Technical report, ORA, June 1990.
- [38] ORA. Romulus: A computer security properties modeling environment, final report - volume 3: Specification languages. Technical report, ORA, June 1990.
- [39] ORA. Romulus: A computer security properties modeling environment, final report - volume 4: Security modeling support. Technical report, ORA, June 1990.

- [40] ORA. Romulus: A computer security properties modeling environment, final report - volume 5: Mathematical component. Technical report, ORA, June 1990.
- [41] Ora. Final report: Task1. Technical Report 91-0003, Ora, 1991.
- [42] Ora. Romulus theories of service assurance. Technical Report 91-0001, Ora, 1991.
- [43] ORA. Romulus, release 2.0, December 1992. Odyssey Research Associates.
- [44] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM Operating Systems Review*, 21(1):8-10, January 1987.
- [45] R. L. Rivest. Cryptography. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, chapter 13, pages 717-755. 1990.
- [46] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [47] R. L. Rivest. The md4 message digest algorithm, 1989.
- [48] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, February 1978.
- [49] David Rosenthal. An approach to increasing the automation of the verification of security. In *Proceedings of Computer Security Foundations Workshop*, pages 90-97, Franconia, NH, June 1988. The MITRE Corporation, M88-37.
- [50] David Rosenthal. Implementing a verification methodology for McCulloch security. In *Proceedings of Computer Security Foundations Workshop II*, pages 133-140, Franconia, NH, June 1989. IEEE.
- [51] David Rosenthal. Security models for priority buffering and interrupt handling. In *Proceedings of Computer Security Foundations Workshop*

- III, pages 91-97, Franconia NH, June 1990. IEEE Computer Society Press.
- [52] David Rosenthal. Modeling restrictive processes that involve blocking request. In *Proceedings of the Computer Security Foundations Workshop VI*, pages 27-38, Franconia, NH, June 1993. IEEE Computer Society Press.
 - [53] Walter Rudin. *Real and Complex Analysis*. McGraw Hill, 1974.
 - [54] M.E. Smid and D.K. Branstad. The data encryption standard: Past and future. *Proceedings of the IEEE*, 76(5):550-559, May 1988.
 - [55] J.G. Steiner, C. Newuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191-202, February 1988.
 - [56] Ian Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, pages 175-183, September 1986.
 - [57] Ian Sutherland. A theory of security for synchronous deterministic state machines. In Volume III of the ORA SAM final report, 1990.
 - [58] Ian Sutherland. Shared-state restrictiveness. ORA Internal Report, July 1992.
 - [59] Ian Sutherland, Stanley Perlo, and Rammohan Varadarajan. Deducibility security with dynamic level assignments. In *Proceedings of Computer Security Foundations Workshop II*, pages 3-8, Franconia, NH, June 1989. IEEE.
 - [60] D. G. Weber. Quantitative hook-up security for covert channel analysis. In *Proceedings of Computer Security Foundations Workshop*, pages 58-71, Franconia, NH, June 1988. The MITRE Corporation, M88-37.
 - [61] D. G. Weber. Formal specification of fault-tolerance and its relation to computer security. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, Pittsburgh, PA, May 1989.
 - [62] M.V. Wilkes. *Time-Sharing Computer Systems*. MacDonald, London, 1968.

***MISSION
OF
ROME LABORATORY***

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.